VoidShare: A P2P Encrypted File Sharing Project Report

Abstract

VoidShare is a lightweight peer-to-peer (P2P) file transfer application that prioritizes privacy. It uses WebRTC DataChannels for transport and end-to-end encryption with AES-GCM, sharing keys via ECDH (P-256). A minimal WebSocket signaling server is used only for initial peer discovery and SDP/ICE exchange, and no file data passes through the server.

Objectives

- To enable direct, encrypted file sharing between two browsers with minimal latency.
- To use a server for signaling purposes only, avoiding server-side file relaying.
- To provide a simple user experience (UX) with features like peer IDs, QR code pairing, and progress indicators.
- To maintain a codebase that is deployable with Docker and easy to self-host.

System Overview

The system consists of two main components:

1. A Next.js 15 frontend that manages the UI, WebRTC connections, and encryption. The frontend uses React 19 and Tailwind CSS.
2. An Express + ws signaling server that brokers 'register' and 'signal' messages. This server is configured with CORS enabled and runs on PORT 4000.

The signaling process includes the exchange of SDP offers/answers and ICE candidates. Once the DataChannel is established, peers exchange public ECDH keys and begin the encrypted transfer.

Technology Stack

- **Frontend:** Next.js 15, React 19, Tailwind, React Toastify, and QRCode.
- **Backend:** Node.js (Express 5) + ws.
- **Crypto:** AES-GCM (256-bit) and ECDH (P-256) using the Web Crypto API.

Detailed Working

1. **Signaling:** A client registers a random peer ID via WebSocket[2] To connect, the initiator creates an RTCPeerConnection and a DataChannel, then sends an SDP offer and ICE candidates to the target peer ID. The server simply relays these messages without storing data.

2. **DataChannel:** After the DataChannel is opened, each peer exports and exchanges its ECDH public key over the DTLS-encrypted channel.
3. **Key Agreement:** Both peers derive a shared secret using ECDH P-256. The sender then generates a new AES-GCM key for the file and encrypts that key with the shared secret.
4. **Metadata:** The sender sends JSON metadata, including the filename, type, size, IVs, and the encrypted AES key.
5. **Transfer:** Encrypted file bytes are streamed in 32 KB chunks. Backpressure is managed by monitoring the DataChannel.bufferedAmount, with a 16 MB cap and 64 KB low threshold.
6. **Completion:** The sender emits _END_ when the transfer is complete. The receiver assembles the chunks, decrypts the AES key using the shared secret, then decrypts and saves the file.

Security Considerations

All WebRTC traffic is DTLS encrypted, and the signaling server only sees the metadata required to connect peers. A new AES-GCM key is generated for each transfer, and the ECDH-derived key is never saved. The project recommends using WSS for signaling and HTTPS for the frontend in production deployments.

Deployment

Dockerfiles are provided for both the frontend and backend. The voidshare-compose.yaml file launches both on ports 3000 (UI) and 4000 (WebSocket). The NEXT_PUBLIC_SIGNALING_SERVER_URL can be set to ws:// for development or wss:// for production. If a proxy is used, it should be configured to forward WebSocket upgrade headers and terminate TLS at the edge.

Testing & Results

Manual testing across different networks has confirmed stable transfers for typical NATs. Buffer management was effective in preventing UI freezes and maintaining responsive transfers. The project notes that without a TURN server, transfers may fail in some restrictive network environments.

Conclusion

VoidShare demonstrates how a compact WebRTC and Web Crypto stack can be used to create a practical, secure, and server-light file sharing application. By separating signaling from transport and using strong client-side encryption, the system ensures that control and data remain with the users.