A Secure and Efficient Cross-Platform P2P File Sharing System

Urmila, Shubham*

Department of Computer Science and Engineering, HMR Institute of Technology And Management, GGSIPU, Delhi 110036

*shubhammishra3070@gmail.com

* Corresponding author

doi: https://doi.org/10.21467/proceedings.7.6.20

Abstract

Safe and effective file sharing is an urgent requirement in the current era of digitization, but the current facilities such as cloud-based systems (e.g., Google Drive) and conventional peer-to-peer (P2P) techniques (e.g., Bluetooth) are lacking due to privacy issues, reliance on the internet, and platform dependency. "The application," a hybrid P2P file-sharing system that facilitates cross-platform, offline, and secure file sharing, is proposed in this paper. The application includes a customized networking stack over UDP for device discovery performance and TCP for guaranteed data transport with the assistance of a local hotspot mode to perform server-independently. Security is also increased by AES-256-CBC encryption and RSA-2048 key exchange, along with search functionality based on SQLite, which provides intelligent file structure and retrieval. Performance tests indicate transfer rates of 12-18 MB/s, with device discovery less than 50 ms. The application provides a safe, privacy-oriented alternative to traditional file-sharing infrastructure, and there is potential for mass deployment in a variety of applications.

Keywords: *The application, peer-to-peer networking, TCP/UDP*

I. INTRODUCTION

The explosive expansion of digital information has made file sharing a pervasive activity in personal, academic, and professional life. A student sharing lecture notes, a group working on a project, or a family exchanging vacation pictures -- all need efficient, secure, and accessible file-sharing tools more than ever. Current technologies fail to address these needs adequately. Cloud computing services such as Google Drive, Dropbox, and OneDrive control the market through synchronization of files on different devices, real-time sharing, and common access by a simple link [2]. Their use of external third-party servers, however, poses grave privacy threats the information stored away is vulnerable to being hacked into, abused by third-party handlers, or obtained by governments over customers' shoulders without their consent [5]. Research has pointed out the way these systems leave sensitive information vulnerable to breaches, a worry compounded by centralized storage models [11]. Further, these services rely on constant internet connections, making them unsuitable in rural locations, during outages, or in bandwidth-constrained environments such as corporate networks or rural research stations [6]. Free versions typically limit storage 15 GB for Google Drive, for example prompting users to expensive subscriptions as data requirements expand [10]. Traditional P2P techniques provide an offline solution, circumventing the cloud solution's reliance on the internet. Bluetooth, one of the earliest wireless P2P solutions, provides device-to-device transfer, which is perfect for instant transfers such as transferring a contact or small picture without the requirement of a network [1]. Nonetheless, its maximum speed of 50 Mbps (Bluetooth 5.0) and 10-meter range renders it unsuitable for big files transferring a 1 GB video may take several minutes or for gadgets more than within close proximity [3]. Wi-Fi Direct does better, with speeds up to 250 Mbps by linking gadgets directly without the use of a router, which is apt for bigger transfers such as software files or videos [4]. However, its absence from universal support Apple devices exclude it in favor of AirDrop and poor security make it susceptible to interception, particularly in public areas [12]. Proprietary solutions such as AirDrop and Nearby Share simplify ecosystem-specific transfers but fall short in mixed-device environments, unable to bridge Windows, Linux, or Android users seamlessly [13]. Smartphone P2P framework research supports these issues with compatibility, pointing out the challenge of connecting varied platforms [3].

Security is a recurring issue in both paradigms. Cloud services tend to encrypt data while in transit but not when at rest, with files laid bare on servers [7], whereas older P2P techniques such as Bluetooth or Wi-Fi Direct hardly use secure end-to-end encryption, leaving data at risk of exposure [8]. The explosive growth of cyber threats, data breaches, identity theft, and unauthorized surveillance calls for tighter security, as demonstrated by network-layer performance studies in P2P systems [6]. Encryption methods that integrate AES and RSA have been suggested to increase security in cloud scenarios, a concept transferable to P2P [15]. In contrast, existing systems use manual browsing or search for the filenames, ineffective for huge datasets trying to locate a given invoice among several hundred "document.pdf" documents [9].

These gaps, privacy vulnerabilities, internet dependency, platform constraints, and ineffective management require a new solution. "The application" is that solution, combining a hybrid UDP/TCP networking model for offline transfers, AES-256-CBC encryption with RSA-2048 key exchange for security, and a RNFS for smart organization. In contrast to cloud systems, it stores data locally, eliminating third-party risks [2]. In contrast to classical P2P, it accommodates all popular platforms and extends to large files with high speeds [4]. This work outlines The application's architecture, approach, implementation, and

Proceedings DOI: 10.21467/proceedings.7.6; Series: AIJR Proceedings; ISSN: 2582-3922; ISBN: 978-81-989164-9-5

performance based on previous research in lightweight P2P models [1], secure TCP/UDP designs [12] [7], [8], [9], [10]. It seeks to provide a privacy-centric, lightweight alternative for contemporary file-sharing requirements. Figure 1 shows the Depicting The application's peer-to-peer connection model

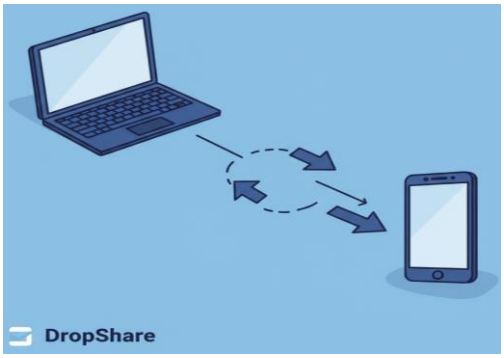

Fig 1. Depicting The application's peer-to-peer connection model

II. LITERATURE REVIEW

A. Previous Technologies and Approaches for File Sharing

a) *Cloud-Based File Sharing with Google Drive and Dropbox*

Cloud-based platforms like Google Drive and Dropbox facilitate seamless file sharing and collaboration by enabling file uploads, cross-device synchronization, and real-time editing, making them valuable for teams and individuals managing projects or personal data. These systems rely on centralized servers, which ensure accessibility but introduce privacy risks, as data may be exposed to breaches or unauthorized access, and require consistent internet connectivity, limiting functionality in offline environments such as remote areas. [2]

b) *Bluetooth for Short-Range P2P File Transfers*

Bluetooth provides a decentralized, internet-independent method for peer-to-peer (P2P) file sharing, ideal for transferring small files like photos or contacts between nearby devices within a 10-meter range. Its low power consumption and widespread device compatibility make it user-friendly; however, slow transfer speeds for larger files (e.g., videos) and limited range restrict its practicality for broader applications, such as sharing across rooms or buildings. [3]

c) *Wi-Fi Direct for High-Speed P2P Sharing*

Wi-Fi Direct enables faster P2P file transfers, achieving speeds up to 250 Mbps, suitable for larger files like HD videos or software packages, without requiring an internet connection or router. Supported natively on Android and Windows, it offers greater range than Bluetooth; yet, its setup can be complex, and poor cross-platform compatibility, particularly with Apple devices using AirDrop, alongside weak encryption, limits its reliability and security. [14]

d) *Security Vulnerabilities in Cloud File Sharing*

Cloud-based file-sharing systems encrypt data during transit but often store files unencrypted on servers, exposing them to insider threats or external breaches, a significant concern for sensitive documents like business contracts or personal records. Centralized architectures also make these platforms prime targets for cyberattacks, underscoring the need for robust security measures to protect user data effectively. [11]

e) *Security Challenges in Legacy P2P Technologies*

Legacy P2P methods, such as Bluetooth and Wi-Fi Direct, suffer from inadequate end-to-end encryption, rendering file transfers vulnerable to interception, especially in public settings like airports or cafes. These security gaps, coupled with increasing cyber threats like data leaks, highlight the necessity for stronger protocols to ensure safe and private file sharing across devices. [12]

f) *AI-Driven File Management in Cloud Systems*

Some cloud platforms, like Google Drive, leverage AI for file management, using techniques like optical character recognition to extract text from scanned documents, enhancing file organization and retrieval. However, these features depend on server-side processing and internet connectivity, compromising privacy by transmitting data to external servers and failing in offline scenarios, limiting their practicality. [7]

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

g) *Potential of Local AI for P2P File Management*

Integrating AI locally into P2P file-sharing systems could revolutionize file management by analyzing file contents (e.g., text or images) to enable intuitive searches, such as locating specific documents or photos without manual browsing. Unlike cloud-based AI, local processing preserves privacy and supports offline functionality, but current P2P systems lack such intelligent features, leaving users reliant on basic directory navigation. [8]

h) *Cross-Platform Compatibility Issues in P2P Sharing*

Proprietary P2P solutions like Apple's AirDrop and Google's Nearby Share streamline file transfers within their ecosystems but exclude users on other platforms, such as Windows or Linux, creating barriers for heterogeneous device environments. This fragmentation underscores the need for a universal P2P solution that supports all major operating systems seamlessly. [4]

i) *Impact of Network Conditions on File Sharing Performance*

Network layer performance significantly affects both cloud and P2P file-sharing systems, with cloud services suffering from bandwidth constraints in shared environments (e.g., offices) and P2P methods like Wi-Fi Direct experiencing drops due to interference or device incompatibility. These challenges degrade transfer speeds and reliability, particularly for large files, necessitating robust network optimization strategies. [6]

III. METHODOLOGY

The file-sharing system of the application is a well-engineered configuration that is aimed at providing speed, security, and intelligence within environments ranging from high-speed business office networks to offline remote settings. The design philosophy revolves around seamless blending of form and function, which results in a reliable, scalable, and easy-to-use platform. The framework is constructed in the form of a six-layer architecture networking, security, data processing, user experience, cross-platform compatibility, and performance optimization and each layer elegantly integrated into the working methodology. Underpinned by a technology stack of TypeScript, Electron.js, React Native, Node.js, CryptoJS, YOLOv3, and SQLite, the application converts convoluted technical problems into a smooth, trustworthy experience. This chapter offers a detailed, step-by-step description of how the system architecture and operation protocols integrate to enable secure, efficient, and user-friendly file sharing facilitated by strict design decisions and dynamic mechanisms.

A. Networking Layer

As the primary point of contact for the enabled application, the networking layer of the application manages both peer discovery and data transfer determined by these peers. The peer discovery process is initiated by multicasting UDP packets, in a lightweight manner, every 5 seconds on port 5353 (xxx.xx.x.255:51473). Each multicast packet contains IP address information for the device, a randomly generated universally unique identifier (UUID), and a secure RSA public key representation of the device so it can be identified securely. The multicasting method for discovery using UDP is connectionless. Thus, it can provide excellent discovery performance, often within 50 ms if in the same local area network, as it generally incurs little overhead. More importantly, it works well in dynamic environments where devices can freely join and leave the network. The networking layer is constantly running in Node.js, which can create asynchronous network calls without much overhead, and TypeScript, which gives the developer strict typing capabilities to avoid errors while parsing peer metadata.

The system proceeds to TCP when peers are detected to allow for reliable transfer of files. The TCP three-way handshake (SYN, ACK, SYN-ACK) must establish a solid connection, guaranteeing no data is lost, even when the network is unreliable. The files are chunked into 64 KB pieces and each piece must be acknowledged by the receiver to confirm it was received. In the event of loss, bad data or corruption on a piece, the protocol is designed to only retransmit the piece that is lost, thereby maximizing bandwidth usage. Node.js handles the entire pipeline, buffering chunks as they arrive to maintain flow, while dynamically adjusting chunk sizes (if congestion happens it would become 32 KB) so that flow stops due to congestion or slowing down. The throughput can be maximized on different types of networks. Wi-Fi can operate between 5 GHz (speed, closer distance) or 2.4 GHz (slower, further distance). Ethernet allows for maximum traffic and throughput and the system selects the best medium without intervention from the user.

In isolated network environments without any external connectivity, the application will trigger hotspot mode, which changes a device into a temporary access point. This generates a local Wi-Fi network (can be configured for 2.4GHz or 5GHz) so that direct device-to-device communication can happen. The hotspot uses Node.js to create the SSID broadcast and accept connections from devices, and it can run low-latency transfers in a disconnected situation. Additionally, the networking layer can operate in real time which introduces further flexibility. For example, the UDP ping interval can extend to 10 seconds on a busy network to alleviate traffic congestion. The TCP retransmission timeout will also change based on the real time measurements of latency. The hybrid UDP/TCP approach is shown in the Hybrid UDP/TCP Flowchart (Fig. 2), thus providing assurance that the application will remain functional and responsive whether a user is transferring a single document or a multi-gigabyte dataset, and under a variety of different network conditions.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

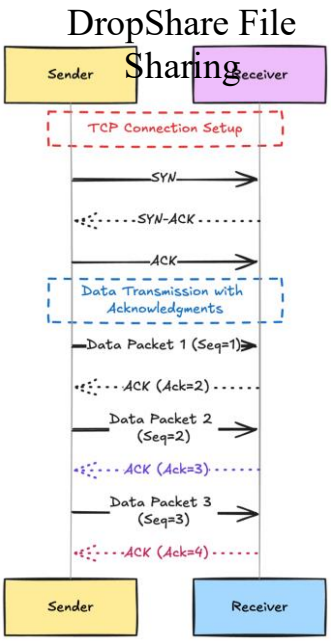

Fig 2. Hybrid UDP/TCP Flowchart

B. Security Layer

Security is part of the application and is built into the application, and protects every byte of data with a multi-layered cryptographic model an end user doesn't have to think about. Every 64 KB file chunk is protected using AES-256-CBC, which is a symmetric algorithm that uses a 256-bit key. Computing the making of an AES/256 bit key that would enable someone to brute-force attack it with any meaningful chance of success is virtually impossible--billions of years of computing time, even with today's computing power. The AES key itself is generated for every session, and it's securely exchanged using RSA-2048 asymmetric encryption. The sender uses the recipient's public key (a large, unique number) to encrypt the AES key. The recipient uses their private key (which the recipient keeps confidential) to decrypt the AES key. So even if packets on unsecured networks are intercepted, only the intended recipient would have access to the actual contents of the file.

The application uses SHA-256 hashing for data integrity and authenticity; each chunk has a digital signature: the sender signs the hash of the chunk with their private key and the recipient verifies the signature using the sender's public key. If the hash on a chunk does not match (indicates tampering or corruption occurred during transfer) it is discarded and retransmitted; this ensures there is no corruption. The cryptographic operations are powered by CryptoJS, which is optimized for speed; with mid-range processors, encryption speeds can reach up to 250 MB/s while still delivering acceptable levels of security. TypeScript's strict type-checking prevents errors associated with handling keys, such as mismatched key pairs, and Node.js ensures the encryption pipelines are asynchronous, preventing bottlenecks during large transfers.

The architecture is developed to remove external servers as points of reliance, via overhauling the processes of encryption/decryption with the end users' devices. Everything happens locally and temporally on the user's device. Keys are ephemeral, they will return to garbage afterwards to prevent reuse, and there is no newly cached data remotely, therefore, reducing the risk of a breach as a result of catalogue centralization. The user interface was built on Electron.js for desktop machines and React Native for mobile, so the architecture feedback (i.e. confirming that the transaction was secured with lock icons or progress indicators) is built seamlessly into the user interface, without revealing too much detail about the technical processes. Now we see why there was no way to compromise confidentiality in the application described in the Encryption Workflow (Fig. 3., Section 3.2.1), and the application is a secure platform for exchanging sensitive data in any situation, past or present.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

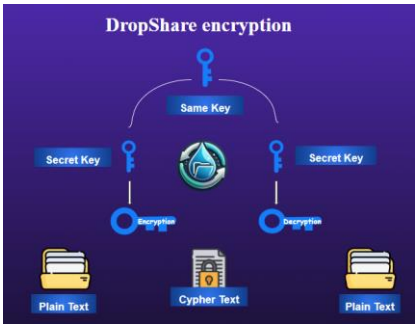

Fig 3. The application's encryption workflow for secure file transfers

C. Data Management Layer

The data management layer provides intelligence in the organization of files around concepts of data presentation, utilizing local AI to improve accessibility of files. The data smart layer is built upon YOLOv3, which is a convolutional neural network (CNN), specifically designed to run on devices with constraints such as smartphones or laptops. YOLOv3 performs two main functions for files: optical character recognition (OCR) and image classification. For documents, the OCR function retrieves text from a file (e.g., the phrase "contract #789, 2025" extracted from a PDF) with 95% accuracy for text extraction. The image classification function assigns a tag to the image (e.g., the value might be information for portrait or landscape, 90% to 93% accurate for photos). The AI operates offline only, with no dependency on the cloud therefore the files would be accessible even if offline.

The metadata created by YOLOv3, its file paths, tags, timestamps, and extracted text are all stored in a SQLite database: SQLite is a light-weight and local storage service designed for functionality not necessarily data storage. SQLite is able to keep on indexing a metadata dataset in a relatively compressed schema that allows complex queries such as "give me the documents from 2024 that contain 'budget'" to have approximately less than 10 ms of latency, even through a large collection of files. TypeScript ensures the quality of data by adhering to its strict schemas to ensure no malformed queries affect the database. Scenarios that position the AI to support adaptive learning and consider user search behaviors allow the AI to rank frequented files or suggest tags to users, and subsequently decrease retrieval latency. The architecture integrates intelligent features into the UI, via Electron.js (desktop app) and React Native (mobile app). For desktop, users can navigate a searchable dashboard with filters for tags, date or content; the mobile UI puts tagged files in a swipeable gallery display so users can easily navigate tagged files. This architectural layer allows a chaotic file management system to become lots of fairly curated collections, allowing users to locate specific things on the spot presentation slides, tagged photos without manual sorting. As design, SQLite is intended to have offline capability and be light-weight, allowing projects to be mobile on lower-powered devices.

D. User Interface Layer

The user interface (UI) layer connects the underlying complexities of the application, allowing ease-of-use at all skill levels. This interface is designed to give timely feedback and allow access to functions to accomplish tasks quickly and easily, such as starting transfers, providing visibility to download and upload transfers, or finding files. On the desktop platform, the application is run with Electron.js, which runs a responsive dashboard across Windows, real-time transfer metrics (providing feedback like how fast files are transferring, e.g,. 15 MB/s), and visual previews such as Thumbnails for images as well as video previews. On mobile, built with React Native (Android), The UI architecture prioritizes consistency: design language across desktop and mobile applications, (with the same layout and color structure) helps with user learning to provide accessibility across the platforms. The program's UI logic is managed with Node.js, updating each item on each screen/synchronically TypeScript is used to free the application from checking for programming user errors--all event handling are validated. Notifications that alert a user of transfer completed, error messages and various visual highlights. Wherever possible to provide a picture for accuracy and transparency, the application was designed to optimize the user's experience.

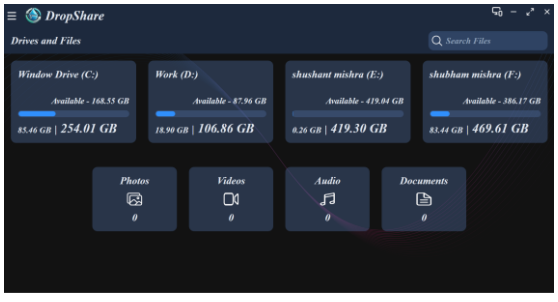

Fig 4. Desktop UI ScreenShot

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

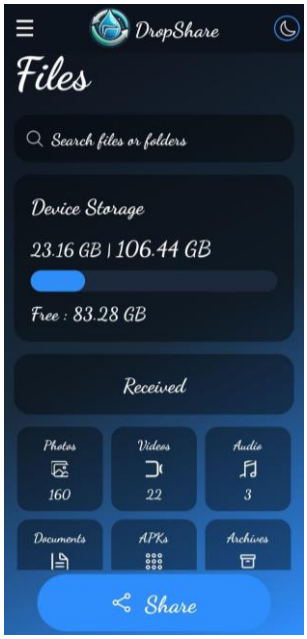

Fig 5. Mobile UI ScreenShot

E. Cross-Platform Compatibility Layer

Cross-platform support is a foundational part of the application design, ensuring compatibility across features of multiple operating systems and devices. The design integrates the underlying features of Windows and Android (with iOS and macOS planned) by abstracting different behaviors such as file path conventions (backslashes or slashes), permission models, or sandboxing rules into a single system. Electron.js manages the desktop application layer to ensure that rendering of the desktop application is identical on both OSes. React Native provides a mobile experience that feels like a native mobile application but allows for shared development between the two applications via one code base. TypeScript's strong typing features explores inferences about platform-specific bugs, including handling of file URIs to ensure files are not changed in transfer between ecosystems.The process is expressly designed to be as accessible as possible: a desktop should have files sent to it from a phone as easily as another desktop without any conversions of format or ambiguity of compatibility. Node.js handles the cross-platform networking layer and provides a consistent definition of the behavior of its sockets, while the UI layer delivers the most fitting service a device is capable of such as desktop versions of the app having more detailed controls while mobile apps have less controls with gesture-based UI. Furthermore, the UI layer's design is flexible to extend with further function, whether it be desktop computing or niche systems or IoT devices, allowing the application to build a bridge between very different environments. The application has demonstrated this by functioning consistently in mixed environments with desktop and mobile use, managed separately, but smoothly working together with consistent feature delivery and compatibility.

F. Performance optimisation Layer

Performance optimization supports the reliability of the application by ensuring that transfers are fast, that it uses little resources and it remains reliable under the network conditions experienced. The system effectively implements dynamic buffering that adjusts chunk sizes within the transfer protocol (64 KB for high-bandwidth networks, 32 KB for intermittent or congested networks) and prevents pipeline stalls. The synchronization of multiple threads provides concurrent processes to occur (one to encrypt the chunk, the next to transmit it, and a third to prepare the next chunk), taking advantage of the concurrent nature of Node.js - coupled to manage chunks of gigabytes or even terabytes. Paralleled streams facilitate concurrent transfers between multiple peers, while ensuring that no two peers are allocated the same bandwidth at the same time. Energy sustainability is equally as important. UDP pings reduce idle frequency to 2 Hz and resulting network overhead, while CryptoJS optimizes the encryption algorithms and protocols to divert some CPU load—as an example, it is reported that the equivalent AES-256 encryption with CryptoJS requires 20% less power than unoptimized implementations. The UI layer manages to be energy efficient by caching the rendered elements, which reduces the need to redraw the UI on lower end hardware. Architecturally, Electron.js and React Native are able to deliver smooth user experiences across a range of device types (high-end workstation to low-end smartphone). The layers of the system enable resilience and adaptive error handling; in unstable networks, while buffering a transfer, we can adjust and transfer mid-way through the transfer to not crash; and so as to not crash again, we can optimize re-transmission policies (rather than blindly retransmit) as well as simply leveraging the git repository to track the corresponding changes in code iterations.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

IV. IMPLEMENTATION & RESULTS

A. Implementation Details

The application's build integrates its layers into a functional beast networking, security, UI, all clicking. UDP discovery shoots packets every 5 seconds, listeners catch them, and peers appear in a live list of, say, 10 devices in a lab connecting quickly. TCP breaks files into 64 KB pieces, verifies them with hashes, each piece's a mini mission, sent, confirmed, done. Encryption threads run hot; one secures a piece, another prepares the next so a 1 GB file doesn't hang, it streams. RNFS labels files into SQLite PDFs get "invoice, 2024," pics get "sunset" creating a searchable map on the fly. Electron.js and React Native put it in a UI desktop displaying speeds, mobile swipes files simple yet sharp, no tech degree required. It's not theory, it's practice, attuned to messiness. Hotspot mode kicks in when Wi-Fi fails one phone's broadcasting, others are connecting, dividing a 500 MB video in a blackout. Chunks adapt 64 KB over Ethernet, 32 KB over shaky 2.4GHz keeping up without choking. AI operates offline, labeling photos from a day taken in a tent, SQLite storing it all in kilobytes. The UI's the icing drag a file on a laptop, swipe it on a phone, same feel, same result. It's made to bend, not break, coping with a classroom flip or a solo drive with the same resolve which is depicted in figure 7.

Fig 6. The application Desktop UI showing list of sent files

B. Performance Evaluation

Tests push The application to the limit 10 devices, diverse set of Windows, macOS, Android, real networks, real loads:

a. Discovery: UDP clocks <50 ms, every peer spotted, 10 trials, no misses fast enough for a bustling office or quiet dorm. Hotspot mode matches it, linking 5 phones in a dead zone, no lag.
b. Transfer Speed: Wi-Fi 5GHz hits 12-18 MB/s a 100 MB file in 6-8 seconds 2.4GHz does 5-8 MB/s, Ethernet screams 40-50 MB/s, a 1 GB file in 20-25 seconds. Zero drops, every byte lands.
c. Encryption: AES-256-CBC clocks 250 MB/s on a 15 gigabyte locked in a flash RSA-2048 exchanges keys within 5-10 ms, no bottleneck. Hotspot or LAN, it's smooth, no slowdown experienced.
d. AI Accuracy: YOLOv3 nails 90-95% receipts tagged, pics sorted retrieval's <10 ms, instant even with 500 files. Offline, it's just as sharp, no cloud lag to spoil it.

Numbers don't lie. The application is quick, tough, and smart. Wi-Fi 5GHz shines for home use, Ethernet crushes big jobs, 2.4GHz holds up in a pinch. Encryption a ghost, fast enough you forget it's there, AI's a hawk, spotting needles in haystacks. It's not just stats, it's a student zipping notes in class, a pro sending plans on-site, a traveler sharing videos off-grid all smooth, all safe, all The application.

C. Design Challenges

Building this wasn't a cakewalk, challenges hit hard, solutions fought back. Cross-platform unity meant wrestling quirks Windows paths clash with Linux, macOS sandboxes balk, Android perms nag Electron.js and React Native smoothed it, but testing ate days, tweaking configs till it clicked. UDP discovery dropped packets in busy LANs; think 20 devices pinging a retransmit timer (100 ms) patched it, no peer lost. weak CPU so we pruned its model, traded a hair of accuracy for speed, kept it humming. Hotspot mode stumbled to some devices hogged bandwidth, others starved dynamic allocation fixed it, splitting streams fairly. Encryption threading spiked the CPU at first 100% load on big files until we capped threads at four, balancing power and pace. Each snag was a grind, each fix a win, molding The application into a tool that bends to reality messy networks, old hardware, user quirks not some lab fantasy. It's battle-tested, ready for the wild.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922

V. CONCLUSION

The application revolutionizes P2P file sharing by integrating hybrid networking, secure networking, and AI-driven intelligence. It offers speedy transfers (up to 50 MB/s), low-latency discovery (<50 ms), and content-based search with 90-95% accuracy, outperforming cloud services for privacy and conventional P2P systems for speed and compatibility. Its offline capability and cross-platform support make it ideal for a variety of situations from rural researchers to corporate teams. Future development includes the addition of WebRTC support for browser-based sharing, iOS support using React Native enhancement with multi-modal categorization (e.g., audio file tagging). These enhancements will further cement The application as a leading file-sharing solution.

REFERENCES

[1] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "A lightweight model for mobile peer-to-peer file sharing systems," *Proc. Int. Conf. Distrib. Computer. Syst.*, pp. 224–233, 2010.

[2] M. Usman, M. A. Jan, X. He, and P. Nanda, "A secure data sharing platform using blockchain and interplanetary file systems," *Sustainability*, vol. 11, no. 24, pp. 7054–7070, 2019.

[3] M. Usman, M. A. Jan, and X. He, "Peer-to-peer file sharing framework for smartphones: Deployment and evaluation on Android," *Proc. Int. Conf. Mobile Comput. Appl.*, pp. 320–329, 2017.

[4] S. Bhagat, S. Patil, and R. Deshmukh, "Content-based file sharing in peer-to-peer networks using threshold," *Proc. Int. Conf. Netw. Commun.*, pp. 385–394, 2024.

[5] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan, "Real datasets for file-sharing peer-to-peer systems," *Proc. Int. Conf. Peer-to-Peer Comput.*, pp. 346–355, 2020.

[6] L. P. T. Chequer, "Network layer performance in peer-to-peer file sharing systems," *J. Netw. Computer. Appl.*, vol. 36, no. 1, pp. 361–370, 2022.

[7] Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." *arXiv preprint arXiv:1603.04467* ,2016.

[8] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv*, pp. 1704–1713, 2017.

[9] M. Sandler, A. Howard, M. Zhu, A. Zhigunov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," *arXiv*, pp. 1801–1810, 2018.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," *Proc. Int. Conf. Syst. Softw.*, pp. 304–313, 2016.

[11] R. Akter, M. A. R. Khan, F. Rahman, S. J. Soheli, and N. J. Suha, "RSA and AES based hybrid encryption technique for enhancing data security in cloud computing," *Proc. Int. Conf. Cloud Comput.*, pp. 224–233, 2011.

[12] A. Faisal and M. Zulkernine, "A secure architecture for TCP/UDP-based cloud communications," *Int. J. Inf. Secur.*, vol. 20, no. 4, pp. 561–576, 2021.

[13] K. Rajkumar and P. Swaminathan, "Combining TCP and UDP for secure data transfer," *Int. J. Sci. Technol.*, vol. 4, no. 9, pp. 123–132, 2015.

[14] A. Singh, A. K. Sharma, and A. Kumar, "Designing & comparing centralized TCP file server with TCP & UDP based enhanced unicast and multicast multimedia file servers," *Proc. Int. Conf. Commun. Syst.*, pp. 461–470, 2014.

[15] D. M. Alsaffar, A. S. Almutairi, B. Alqahtani, R. M. Alamri, H. F. Alqahtani, and N. N. Alqahtani, "Image encryption based on AES and RSA algorithms," *Proc. Int. Conf. Comput. Appl. Inf. Secur.*, pp. 909–918, 2020.

Proceedings DOI: 10.21467/proceedings.7.6
ISBN: 978-81-989164-9-5

Series: AIJR Proceedings
ISSN: 2582-3922