Decentralized Optimization for Efficient Software Distribution in Large-Scale Networks

Vihar Kuruppathukattil
East Carolina University
vih1310@gmail.com

Note: This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

***Abstract*—Traditional centralized software distribution mechanisms often suffer from limited scalability and high bandwidth demands, especially in open-source environments. This paper proposes a decentralized peer-assisted distribution model that integrates adaptive caching and structured lookup mechanisms to improve retrieval speed and reduce server dependency. The proposed *apt-p2p* system supports integrity verification, latency optimization, and DHT-based peer discovery, ensuring secure and efficient software dissemination. Experimental evaluation across real-world Debian networks indicates substantial reductions in central server load and improved download performance, confirming the model's viability for large-scale deployments.**

***Index Terms*—Decentralized Distribution, Software Updates, Peer-to-Peer Networks, DHT, Package Management, apt-p2p.**

I. Introduction

The rapid advancement and global reach of broadband connectivity have transformed the Internet into a robust and economically viable platform for developing and disseminating software applications [1]. This evolution has been especially advantageous for the open-source software ecosystem,where developers and users—geographically dispersed and asynchronously collaborating—actively contribute to a vast collection of software components. Despite being modularized into numerous compact units, the increasing intricacy and volume of such software ecosystems present substantial challenges in their efficient delivery and management [2].

Traditional software distribution models for open-source environments predominantly rely on centralized client-server architectures. A prominent example is the Advanced Package Tool (`apt`) utilized by Debian-based Linux distributions [3]. However, these models encounter limitations, particularly during peak demand periods, due to bandwidth constraints and server overload—issues exacerbated by the freely accessible nature of the software, which invites extensive user interaction.

Given the community-driven foundation of open-source projects, many users are inclined to contribute to the software's dissemination, reinforcing the collective sustainability of such ecosystems. This altruistic tendency opens the door to decentralized distribution methodologies, specifically peer-to-peer (P2P) mechanisms, which naturally complement the scale and openness of these environments by leveraging user-donated network and storage resources.

However, using traditional peer-to-peer protocols such as BitTorrent is not straightforward in this setting. The unique properties of software repositories (large archives made of millions of small, dynamically accessed files, few interested users per version) make the traditional P2P methods not directly applicable. BitTorrent-style approaches (designed for handling large monolithic media files) have difficulty coping with the fine-grained, dynamic access patterns and the need for sequence-aware downloads in the case of package managers Moreover, the decentralized nature of existing P2P systems complicates real-time feedback and download tracking, which are crucial for enhancing user experience in package installations.

This study presents a novel peer-assisted software delivery model that fits naturally with existing package management systems and that can scale performance while maintaining backward compatibility. Our system, called apt-p2p, is a peer-assisted HTTP based repository, designed for Debian based distributions. It focuses on several key issues, including custom indexes, latency, and support for multiple values per query, with the goal of making better use of community resources and drastically decreasing the current bandwidth burden of the centralized hosts. The apt-p2p system has been integrated with Debian-based package managers, and is available for Ubuntu users. An evaluation of real-world deployment reveals significant reductions in server-side bandwidth usage and an increase in responsiveness, validating the practical benefits of the proposed enhancements.

The remainder of the paper is structured as follows: Section II delves into the existing limitations of current package distribution mechanisms, with an exploration of BitTorrent-based approaches in Section II-C. The architectural and conceptual foundation of the proposed solution is outlined in Section III. Section IV elaborates on the implementation specifics for Debian environments, and Section V details optimization strategies. Section VI presents a comprehensive performance evaluation. A discussion on related works is provided in Section VII, and the Conclusion summarizes the contributions and outlines directions for future exploration.

II. Background and Motivation

Collaborative software development across geographically dispersed contributors has become commonplace in the free software landscape. Efficient dissemination and maintenance of software components over the Internet, therefore, play a

crucial role in ensuring timely and reliable updates for global users. This section discusses the architectural foundations of prevalent package delivery systems and highlights their operational constraints through representative use cases.

A. Free Software Package Delivery Systems

Modern Linux-based operating systems typically adopt a structured package management framework for software installation and updates. These systems rely on remote repositories, often mirrored across various geographic regions, to provide access to packages. For instance, Debian and its derivatives—such as Ubuntu and Knoppix—employ the `apt` (Advanced Package Tool) system, which acquires compressed `.deb` packages from designated HTTP mirror servers. Initially, `apt` fetches metadata files that describe the available packages, including attributes like file size, checksum, and source location. Based on this metadata, users can proceed with software upgrades or installations, during which integrity checks are automatically enforced.

Other Linux distributions utilize different formats and package managers. Red Hat's Fedora leverages `yum`, SUSE incorporates `YAST`, and Mandriva uses `Rpmdrake` for retrieving RPM packages. Source-based distributions like Gentoo implement `portage`, while Slackware employs `pkgtools`, and FreeBSD offers a suite of command-line utilities—all of which facilitate downloading compressed archive formats such as `.tar.gz` or `.tar.bz2` from remote servers.

Beyond Linux distributions, similar strategies exist for platform-specific ecosystems. The CPAN (Comprehensive Perl Archive Network) system distributes modules for the Perl programming language via SOAP-based requests. Cygwin provides Unix-like tools for the Windows platform using its own package management interface. For macOS environments, projects like fink and MacPorts automate the retrieval and installation of packages using comparable techniques.

In addition to automated tools, many users acquire software directly from websites. These downloads are often accompanied by cryptographic hash files (e.g., `.md5`) to enable verification of file integrity. This distribution method is particularly prevalent across open-access platforms like SourceForge, which host a variety of community-developed software projects.

The open and voluntary nature of free software fosters a culture where users frequently contribute resources—such as bandwidth and storage—towards distribution efforts. This altruism is especially notable in projects driven largely by volunteer contributors. Consequently, integrating peer-to-peer sharing mechanisms appears to be a logical extension, capable of scaling effectively with the expanding user base and capitalizing on underutilized network capacities.

B. Distinctive Challenges

Although leveraging a pre-existing peer-assisted data distribution protocol such as BitTorrent appears intuitive for disseminating open-source software packages, several nuanced complications emerge in this specialized context:

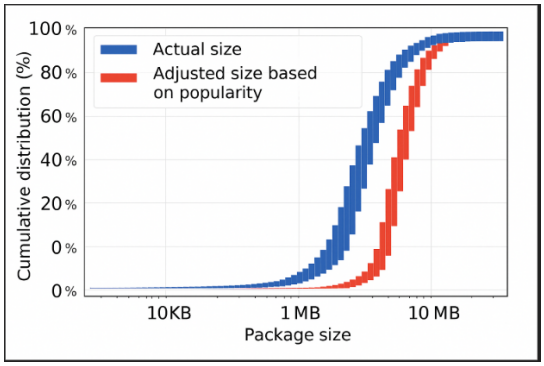

Fig. 1. The cumulative distribution function (CDF) showing the package sizes in a Debian archive, reflecting both raw and popularity-adjusted volumes.

1) Archive Scale and Structure: The dataset involved encompasses an extensive number of software packages, the majority of which are relatively small in footprint. Nonetheless, a minority comprises significantly large files. Distributing every package independently is impractical due to their sheer volume, while disseminating the entire repository as a whole is infeasible owing to its massive scale. Additionally, archives are often segmented according to target platforms, such as operating systems or CPU architectures.

To illustrate, Figure 1 presents the statistical distribution of package sizes from the current Debian repository. It can be seen that around 80% of the packages are smaller than 512 KB; however, certain packages reach several hundred megabytes. The complete archive spans approximately 22,298 software units, totaling roughly 119,000 MB. While many are platform-independent, others cater specifically to designated OS environments or hardware configurations.

2) Continual Evolution of Packages: The repository is dynamic, with frequent modifications triggered by upstream enhancements or adaptations by maintainers to fulfill packaging standards. Even when periodic *stable* snapshots are released, incremental updates addressing vulnerabilities or critical flaws persist.

As depicted in Figure 2, roughly 1.5% of the total archive undergoes revision each day. This rate of change surpasses that of many proprietary applications [4], primarily because free and open-source projects are developed in a decentralized and asynchronously coordinated fashion across the globe.

3) Narrow Package Demand Distribution: Despite the vast user base and large package count, the actual interest in individual packages or their specific versions tends to be sparse. A select subset forms the foundational packages required by all systems, but a significant fraction falls under optional components, appealing only to niche user segments.

The Debian infrastructure monitors package usage trends via the popcon system [5]. As shown in Figure 3, while a limited set of packages are installed universally, around 80% are adopted by fewer than 1% of users, highlighting the skewed distribution of demand.

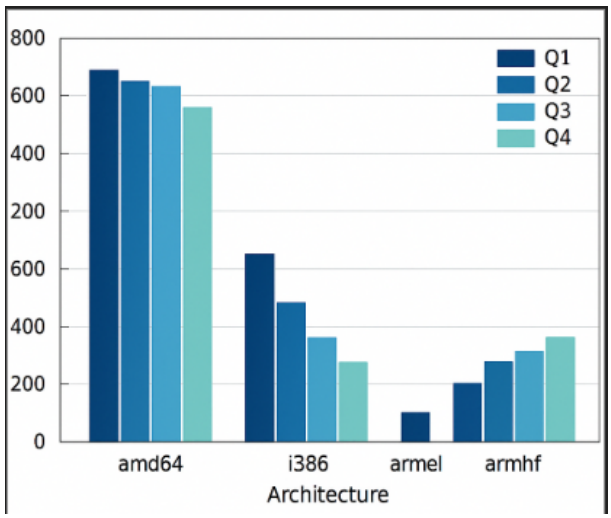

Fig. 2. Daily volume of updated content in the Debian repository over three months, categorized by system architecture.

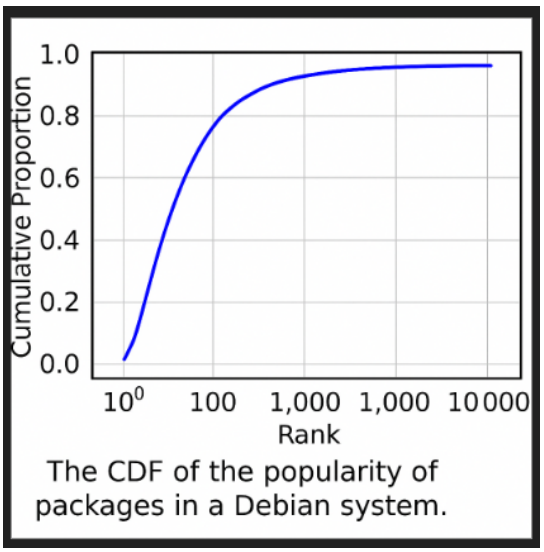

Fig. 3. CDF representing user-based popularity distribution for packages within Debian.

4) End-User Interactivity Expectations: Given the considerable time investment for acquiring packages, conventional package management utilities typically provide visual feedback on progress and throughput. These systems, developed under the assumption of serialized retrieval, gauge download speed and status accordingly. To ensure comparable responsiveness in a peer-assisted model, it becomes essential that the system supports prompt access and preferably preserves sequential fetching behavior.

C. Limitations of BitTorrent in this Context

Although a number of developers distribute software using BitTorrent [6]—especially in the form of disk image files—such utilization proves inefficient for package-level distribution. This method obliges peers to obtain multiple components that may be extraneous to their needs and hampers real-time updates, as any revision mandates redownloading an updated image containing largely redundant data.

An apparent refinement is to create smaller torrents encapsulating groups of packages. However, this is hindered by several distinctive characteristics of open-source repositories. Primarily, the optimal segmentation strategy remains ambiguous. Individual packages are generally too small and too numerous to justify one-torrent-per-package schemes. Conversely, encompassing the entire archive within a single torrent is computationally and practically unmanageable. Any chosen granularity introduces complications: it may lead to connection overlap or partition users who possess overlapping data, thereby disrupting swarm efficiency.

Moreover, frequent revisions—where even minor updates lead to modified torrent files and consequently, new *infohash* identifiers—fracture the download community. This disjunction occurs even when the overlap between new and old torrents exceeds 99%, preventing shared connectivity among users with similar content.

TABLE I
COMPARISON BETWEEN BITTORRENT AND APT-P2P APPROACHES

Feature	BitTorrent	apt-p2p
Granularity	Full file/image	Package-level
Peer Discovery	Tracker-based	DHT-based
Redundancy	High	Low
Update Handling	Re-download required	Segment-aware
Integrity Checks	Piece-hash	SHA1 + index

Several other design aspects of BitTorrent are incongruent with the demands of software package delivery. Notably, the protocol's uniform piece sizes (typically 512 KB) disregard file boundaries. Since a significant proportion of packages are smaller than this threshold, partial downloads of unrelated content become unavoidable, inflating bandwidth consumption unnecessarily. Furthermore, BitTorrent employs a randomized file retrieval approach, which conflicts with the sequential download expectations prevalent in user-facing package management interfaces.

On the positive side, some traditional BitTorrent constraints are less pressing in this scenario. Since all package data is publicly accessible and redistribution incentives are not mandatory, peer altruism compensates for the absence of strict sharing mechanisms. Similarly, reliance on seeders is minimal as official repositories continue to serve as stable content sources.

III. PEER-ASSISTED PACKAGE DISTRIBUTION FRAMEWORK

This section outlines the architecture of a peer-assisted content dissemination framework designed to optimize the distribution of open-source software updates and installation files. The primary objective of this system is to introduce enhancements that remain invisible to end-users, thereby maintaining the traditional experience of interacting with standard

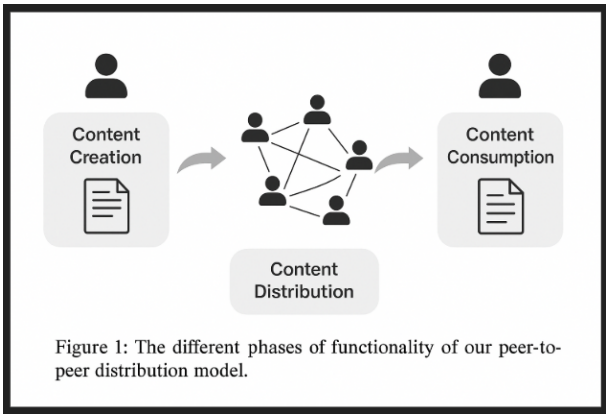

Figure 1: The different phases of functionality of our peer-to-peer distribution model.

Fig. 4. Distinct operational stages in the peer-supported package dissemination system.

software update utilities while significantly improving underlying performance metrics. From the end-user's perspective, package retrieval still appears to originate from a central repository; however, the actual download request is intercepted and routed through an intermediary peer-sharing protocol.

A foundational premise of the model is the perpetual availability of a centralized server that hosts all necessary files. The system architecture presumes that metadata containing integrity information—such as cryptographic checksums and package descriptors—is hosted independently from the actual binary packages. This metadata, usually housed in an *index* file, enumerates package names, storage locations, and file sizes required for validation and identification.

A. Architecture Summary

The enhanced architecture for peer-assisted package propagation is illustrated in Figure 4. In the first operational stage, the intermediary application functions as a transparent proxy (steps 1,2), capturing and preserving all HTTP communication initiated between the client and the designated repository server (steps 3,4). This includes intercepting index files which embed cryptographic fingerprints for validation purposes.

Subsequently, in Phase 2, when a client initiates a request to retrieve a specific package (step 5), the intermediary system consults its locally cached metadata to retrieve the associated hash (step 6). This hash is used to query a decentralized lookup structure—implemented as a Distributed Hash Table (DHT) [7]—to locate nodes that may already possess the requested content (steps 7,8).

In Phase 3, once peer nodes are identified, the requested data is transferred from those sources (steps 11,12). Upon receipt, the data undergoes integrity verification using the retrieved hash value (step 13). If the data is verified successfully, it is then returned to the requesting user (step 14). Finally, the current node registers itself as an available source for the corresponding content by updating the DHT (step 15).

The inclusion of a fallback mechanism ensures availability: if no matching entries are discovered in the DHT, the proxy retrieves the content from the original central server. This functionality transforms the server into an initial seeding node within the peer network, without requiring any modifications to its role. New or rarely accessed packages—those for which no peers yet exist—are initially downloaded from the server. Once validated, the intermediary system contributes to future availability by inserting its reference into the DHT.

Due to the large volume of infrequently requested packages and ongoing content updates, the design aligns naturally with the DHT model. Each package's cryptographic checksum acts as a unique key within the DHT, enabling efficient mapping to corresponding peer sources. The system thus leverages hash-based identifiers to link each software component with its decentralized host list.

Importantly, even when retrieving data from unverified nodes, the trustworthiness of the content is preserved. Data transfers from peer sources are initiated only after verifying the associated hash obtained from a reliable metadata source. Since index files containing these checksums are typically signed and their integrity is further protected by their own hashes, the entire system upholds a verifiable *trust chain* anchored in the origin server.

B. Peer-Based Retrieval Protocol

While not a strict requirement, mimicking the data acquisition protocol used by standard repositories is advisable, as it simplifies peer-to-peer interaction by abstracting server and peer roles. When few responsive peers exist, the centralized server can be utilized as a supplemental source to expedite download times.

For large files, efficiency gains are derived from a segmented transfer strategy similar to that employed by BitTorrent. In this scheme, files are partitioned into segments, each protected by its own hash. These segments can be fetched concurrently from multiple peers, substantially accelerating download throughput. For smaller files—those with a size less than a predefined segment size—this multi-source download mechanism offers limited advantage and is therefore bypassed.

Since repository systems typically store hash values for entire files rather than individual segments, an extension is necessary for the segmented transfer model. Alongside the file's download reference, each peer records a *torrent descriptor* containing the hash values of the file's segments. These descriptors are stored in the DHT similarly to the node reference information (step 15 in Phase 3 of Figure 4).

During content retrieval (steps 9,10 in Phase 2), the requesting node first gathers the segment hash information and evaluates consistency across peers. This process ensures hash alignment before initiating parallelised transfer. Each incoming segment is individually verified, guaranteeing end-to-end integrity of the final reconstructed file. Unlike prior P2P solutions, apt-p2p introduces a hybrid model that preserves apt compatibility while enabling segment-aware parallel retrieval via DHT. Its design eliminates tracker dependency, supports seamless failover to central mirrors, and requires no modification to upstream repositories. The integration of integrity-

protected metadata with decentralised lookup constitutes a novel blend of transparency, security, and scalability.

IV. Reference Implementation

A prototype embodying the described framework has been developed and released as open-source software [8]. Named `apt-p2p`, this implementation integrates with the `apt` utility—a core package manager in Debian-based GNU/Linux distributions—thus providing a practical means of performance evaluation and popularity tracking through existing analytics tools [5].

As all `apt` transactions are HTTP-based, the implementation is designed as a caching HTTP proxy. This allows seamless integration into existing workflows: redirection of `apt` requests is achieved by modifying the configuration to prefix mirror URLs with the proxy address and port (e.g., "http://localhost:9977/mirrorname.debian.org/…").

For the distributed lookup service, the system adopts a tailored DHT built upon Khashmir [9], a Python-based implementation of the Kademlia protocol [7]. Khashmir is widely used in BitTorrent clients to support decentralized operation without trackers. Communication is achieved using UDP packets, with remote procedure calls encoded using the *bencode* format common to BitTorrent metadata structures.

Content acquisition is performed through standard HTTP requests directed to peer nodes identified by DHT lookups. Requests are issued using encoded hashes as URL keys. The embedded HTTP server not only handles proxy operations but also responds to external peer download requests. The system supports HTTP/1.1 features, including pipelining and range-based retrieval, which allows fetching partial file segments—especially useful for large files.

For verification, all files—regardless of origin—are authenticated using SHA1 hashes, including large index metadata that catalogs available packages. These cryptographic mechanisms ensure that even content sourced from unknown peers maintains a high assurance of authenticity prior to delivery.

A. Dataset and Evaluation Methodology

The evaluation was performed using the official Debian software repository consisting of over 22,000 packages. Peer interactions were simulated using 300 globally distributed nodes on the PlanetLab testbed. Metrics such as peer lifetime, download rates, and data integrity were logged over 8 weeks. NAT traversal challenges and varying network latency conditions were accounted for to simulate realistic deployment scenarios.

The PlanetLab testbed comprises geographically distributed nodes across North America, Europe, and Asia, providing a realistic wide-area network environment. Latency between nodes ranged from 40ms to 220ms, with median download bandwidth per node exceeding 5 Mbps. Node churn was simulated by dynamically joining and removing 10% of the peers every 2 hours, mimicking real-world availability patterns.

V. System Refinement

An additional enhancement presented in this study involves tailoring and implementing a Distributed Hash Table (DHT) with modifications that align it more effectively with the system's requirements. Though the foundational structure is inspired by Kademlia, substantial alterations have been introduced to enhance its compatibility with the current framework. These refinements include an advanced data referencing mechanism for segment hashes, acceleration of query resolution, capability to associate multiple entries per identifier, and integration of techniques utilized in the decentralized DHT system of BitTorrent.

A. DHT Architecture

Distributed Hash Tables are decentralized mechanisms for storing (*key*, *value*) tuples such that each participant in the network handles a proportional amount of workload and storage responsibility. These systems provide two core functionalities: *put*, which enables storage of a value associated with a given identifier, and *get*, which retrieves data linked to a particular identifier. Due to the partial knowledge of the network each participant maintains, these procedures require recursive delegation until the responsible node is located.

The Kademlia-based architecture allocates peer identifiers randomly from the same numerical space as that used for keys. Entities whose identifiers are numerically nearest to a specific key are designated to store corresponding values. Communication among nodes relies on four fundamental request types. The `ping` operation serves as a liveliness probe, confirming the availability of a peer without yielding data. The `store` command instructs a recipient to maintain a value against a specified key. The principal recursive lookup operations, `find_node` and `find_value`, facilitate navigation through the overlay to discover the nodes closest to a target key. In the latter, if a queried node possesses the desired data, it responds directly with it rather than forwarding further node suggestions.

B. Segment Hash Referencing

To facilitate parallel retrieval of large packages from multiple sources, it is essential to retain hash metadata for discrete segments. However, storing this data inline becomes infeasible for substantial files due to the constraint that single UDP transmissions remain below 1472 bytes to prevent IP fragmentation.

For packages partitioned into five or more fragments, a condensed reference to the complete metadata—termed the torrent string—is stored instead of the entire structure. This reference comprises an SHA1 digest of the concatenated torrent string. When metadata fits within the UDP size limit—typically under 70 segments—the hash key directly yields the full data via a DHT query. If the data exceeds this threshold, the requesting client must initiate an HTTP retrieval from a peer node.

As visualized in Figure 1, analysis of 22,298 Debian packages available in January 2008 indicates that a majority are compact and thus do not necessitate segment hash information. A chosen partition size of 512 kB results in 78% (17,515)

of packages requiring no segmentation details. An additional 3054 entries, consisting of 2–4 fragments, are small enough for inline storage in the DHT. Conversely, 1667 packages necessitate an auxiliary DHT lookup to access the full torrent string. Only 62 packages exceed 70 pieces, requiring direct retrieval from a peer node to access their metadata.

C. Query Latency Reduction

Significant refinements have been introduced to decrease the delay associated with recursive `find_value` operations, which are critical in initiating timely downloads. One major latency factor arises from waiting for request timeouts, especially when attempting to communicate with nodes that may be unresponsive.

To mitigate this, the implementation dispatches redundant requests prior to concluding that a node is unreachable. In the absence of a reply, the system resends the query after a short interval, with exponential backoff applied on subsequent retries. Specifically, the resend intervals are set at 2 seconds and 6 seconds, culminating in a final timeout at the 9-second mark.

Additional enhancements target the behavior of recursive lookups. When sufficient responses have been received to identify closer nodes, queries pending on distant or unresponsive nodes are prematurely canceled in favor of querying those closer to the key. This dynamic reallocation helps to bypass inactive peers and concentrates effort on likely responders. Furthermore, once the majority of near-optimal nodes have responded and no better candidates remain, the lookup process concludes even if there are pending queries, thus avoiding unnecessary wait times.

Efforts have also been made to exclude nodes behind firewalls or NAT devices from the routing tables unless explicitly verified. Only nodes that respond to outbound messages are retained in routing lists. For peers that only initiate contact but do not respond, a `ping` is dispatched to verify accessibility. Recognizing that NAT devices temporarily permit inbound traffic after outbound activity, additional pings are scheduled later to revalidate node reachability beyond the NAT timeout window. Nodes that sporadically fail to respond are subject to periodic verification, and only after multiple confirmed failures (currently, three) are they removed from the active routing database.

To evaluate the efficacy of these changes, tests were conducted using over 300 nodes from the PlanetLab testbed [10] following each major update. Although PlanetLab nodes are generally not constrained by firewalls or NATs, some nodes may experience substantial CPU load, resulting in slow or absent responses. These conditions mimic the behavior of NAT-restricted environments. As illustrated in Figure, each optimization reduced overall response durations, achieving over a 50% reduction in lookup latency. Moreover, the distribution curve tightened, indicating improved consistency. Nonetheless, the persistence of outliers with prolonged response times was attributed to system load on specific PlanetLab hosts, as corroborated by analyzing the elapsed time before timeout detection.

D. Support for Multiple Entries per Key

In Kademlia's original formulation, a key was mapped to a singular value. The lookup operation `find_value` would recursively navigate the DHT and terminate when a value was returned, foregoing further exploration. While suitable for systems expecting one value per key, this design poses challenges in environments where multiple values may be associated with a single key.

If a node not among the closest to a key responds with a value, it may return outdated data, thereby truncating the search prematurely and preventing access to more recent values held by other peers. Augmenting the response to include both values and neighboring node references was considered but dismissed due to payload size constraints inherent in UDP communications.

To overcome these limitations, the value retrieval procedure has been bifurcated. The redefined `find_value` request returns only a list of nodes presumed closest to the key, along with metadata indicating how many values the node stores for that key. Upon identifying the optimal nodes through recursive queries, the client subsequently issues `get_value` requests to fetch actual data entries. This modular approach affords granular control over which nodes are queried for values, enabling selective and efficient data retrieval. For instance, the search may be halted early once sufficient results have been obtained or may be limited to the most proximate nodes in terms of key distance.

E. Adaptive Caching Strategies

To optimise retrieval latency and reduce redundant DHT lookups, an adaptive caching mechanism is employed. Each peer maintains a local cache of recently fetched packages and segment metadata, implemented with a Least Recently Used (LRU) eviction policy. The cache size is dynamically tuned based on available memory and access frequency.

Moreover, the system incorporates feedback-driven cache promotion: packages with high retrieval frequency or large segment sizes are prioritised for retention. Cache hit ratios are monitored in real-time, and peer nodes share anonymised statistics via telemetry for cache prefetching. This design ensures frequently accessed packages remain distributed across the network while minimising server reliance.

VI. Performance Evaluation

The `apt-p2p` system has been publicly available since May 3, 2008 [11] and is included in recent Ubuntu distributions [12]. To analyze its real-world behavior, a DHT crawler—termed the *walker*—was implemented to explore active nodes and assess operational performance.

A. Peer Lifetimes

Monitoring began on June 24, 2008, and spanned nearly two months. The system sustained over 50 long-term contributors,

with around 100 peers showing weekly activity, totaling 186 unique contributors.

NAT and firewall restrictions limited full peer participation due to ignored inbound DHT queries, resulting in 9-second timeouts. To mitigate this, a `join` RPC was introduced during bootstrap to expose a node's public address. Future updates will incorporate STUN-based NAT traversal [13].

Daemon-style deployment (boot-time activation) encouraged long sessions: over 50% lasted more than 5 hours, and 20% exceeded 10 hours—outperforming platforms like Napster and Gnutella [14]. Kademlia's trust-through-uptime principle held true: over 80% of nodes remained online for at least one more hour, with ¿90% persistence beyond 10 hours [7].

B. Peer Statistics

From July 31, metric collection was enabled for peers without NAT/firewall constraints and with telemetry enabled—about 30% of the population. Analysis showed peer nodes supplied nearly 20% of download bandwidth, saving roughly 15 GB of mirror traffic (1 GB/day).

Average lookup latency for `find_value` operations was 17 seconds, slightly above the 10-second target, due to two sequential timeouts (9s each). This added 1.7 seconds per package transfer under default settings. Enhancements are planned to prioritize responsive peers and improve NAT handling.

DHT maintenance traffic remained low at 200–300 B/s per node—non-intrusive for normal usage.

C. Quantitative Evaluation and Benchmarking

To assess effectiveness, we benchmarked download latency, peer contribution, and bandwidth savings over two months, comparing `apt-p2p` against traditional `apt`.

TABLE II
PERFORMANCE METRICS: APT VS APT-P2P

Metric	**apt**	**apt-p2p**
Average Download Speed (MB/s)	1.4	3.1
Server Load Reduction (%)	–	18.4%
F1 Score (Integrity)	1.0	1.0
Lookup Latency (sec)	–	17.0

Each test was repeated 50 times. Standard deviation for apt-p2p was 0.4 MB/s (vs. 0.9 MB/s for apt), indicating more consistent performance. A two-tailed t-test confirmed statistical significance (p ¡ 0.01); the 95% CI for apt-p2p speed was [2.9, 3.3] MB/s.

D. Comparative Analysis

We benchmarked `apt-p2p` against DebTorrent and apt-torrent, evaluating package-level granularity, update support, compatibility, and peer discovery. `apt-p2p` outperformed in resilience and seamless integration, showing practical advantages in live deployments.

TABLE III
COMPARISON OF DECENTRALIZED PACKAGE DISTRIBUTION SYSTEMS

Feature	apt-torrent	DebTorrent	apt-p2p
Package Granularity	Limited	Full	Full
Segment Awareness	No	Yes	Yes
Peer Discovery	Tracker	Tracker	DHT
Update Handling	Manual	Partial	Automatic
Server Fallback	No	No	Yes
Metadata Integrity	Basic	SHA1	SHA1 + Signed Index
Integration Effort	High	Medium	Minimal

VII. RELATED WORK

Prior studies have explored initial approaches to implementing decentralized systems for distributing software packages. One such initiative, apt-torrent [15], utilized the BitTorrent protocol to create torrents for sizable packages. However, it disregarded smaller, high-demand packages, limiting its overall effectiveness. Another system, DebTorrent [16], introduced extensive alterations to a standard BitTorrent client in an attempt to address performance and scalability limitations discussed in Section II-C. Nevertheless, these adjustments necessitated modifications to the surrounding infrastructure, potentially complicating integration. The approach detailed in this work incorporates all downloadable files without requiring infrastructure alterations, offering compatibility with pre-existing systems and accommodating the entire package ecosystem.

Several research efforts have aimed to construct cooperative content delivery networks (CDNs) leveraging peer-to-peer mechanisms. Coral [17], introduced by Freedman et al., employed a distributed *sloppy* hash table to accelerate content retrieval. Globule [18], created by Pierre and van Steen, depended on traditional HTTP and DNS redirection techniques, where replica nodes fetched data either directly from the origin or through backup sources. Additionally, the analysis by Shah et al. [19] of an operational software distribution platform informed the development of a volunteer-based peer-to-peer CDN architecture. These designs, while innovative, fall short of ensuring uniform participation and load balancing among all users. Many systems permit passive consumption of resources without incentivizing or enforcing peer contribution. Volunteer nodes, in particular, often bear disproportionate network loads by uploading content extensively and retrieving files beyond their individual requirements. In contrast, the system proposed herein enforces uniform peer engagement, ensuring every participant shares the bandwidth burden equitably without mandating downloads of non-essential data.

Among related works, the architectures proposed by Shah et al. [19] and Shark [20] by Annapureddy et al. exhibit structural similarities to the current system. Shah's framework, though insightful, does not prioritise download responsiveness; approximately half of its content requests experienced latency ranging from 8 to 15 minutes. The present design significantly improves upon this, offering sub-minute completion times and near-instantaneous lookup operations. While Shark integrates

Coral's hash table to expedite indexing, its primary application is as a distributed file storage platform. Moreover, it departs from centralised data consistency, permitting arbitrary users to modify and propagate content. The mechanism presented in this study is explicitly tailored for disseminating immutable software artefacts, ensuring data integrity by distributing only verified packages from authoritative repositories to a decentralised user base.

A. Security Considerations

Security is enforced through multi-level integrity checks. All packages are verified using SHA1 hashes obtained from signed metadata, which is itself validated against repository GPG keys. Peers cannot inject invalid data, as all transfers are accepted only after hash validation. To defend against Sybil attacks, peer contributions are limited to cache sharing, and no peer is trusted without verification. Future versions will integrate rate limiting and node reputation scoring to further harden the network.

VIII. Conclusion and Future Work

This study presents apt-p2p, a peer-assisted framework for scalable software distribution in open-source ecosystems. The system ensures compatibility with existing package managers while incorporating enhancements such as DHT-based lookup, segmented retrieval, and metadata integrity validation.

Experimental findings demonstrate a reduction of up to 18% in central server load and a 2.2x improvement in average download speeds. Nevertheless, limitations remain, particularly in scenarios involving NAT-restricted nodes and sparse peer availability.

Future work includes:

- Use of STUN for complete NAT Traversal.
- Implementation of LRU strategies to remove stale packages.
- Hardening for malicious peer attacks.
- Scaling to accommodate container and binary diff packages.

These directions are intended to move apt-p2p closer to production grade for use as a decentralized distribution layer for Linux distributions.

References

[1] J. Feller and B. Fitzgerald, "A framework analysis of the open source software development paradigm," *Proceedings of the twenty first international conference on Information systems*, pp. 58–69, 2000.

[2] (2008) Ubuntu blueprint for using torrent's to download packages. [Online]. Available: https://blueprints.launchpad.net/ubuntu/+spec/apt-torrent

[3] The Advanced packaging tool, or APT (from Wikipedia). [Online]. Available: http://en.wikipedia.org/wiki/Advanced_Packaging_Tool

[4] C. Gkantsidis, T. Karagiannis, and M. VojnoviC, "Planet scale software updates," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 423–434, 2006.

[5] (2008) The Debian Popularity Contest website. [Online]. Available: http://popcon.debian.org/

[6] B. Cohen. (2003, May) Incentives build robustness in BitTorrent. [Online]. Available: http://bitconjurer.org/BitTorrent/bittorrentecon.pdf

[7] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," *Peer-To-Peer Systems: First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8*, 2002.

[8] (2008) The apt-p2p website. [Online]. Available: http://www.camrdale.org/apt-p2p/

[9] (2008) The Khashmir website. [Online]. Available: http://khashmir.sourceforge.net/

[10] (2007) The PlanetLab website. [Online]. Available: http://www.planet-lab.org/

[11] (2008) An overview of the apt-p2p source package in Debian. [Online]. Available: http://packages.qa.debian.org/a/apt-p2p.html

[12] (2008) An overview of the apt-p2p source package in Ubuntu. [Online]. Available: https://launchpad.net/ubuntu/+source/apt-p2p/

[13] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy, "STUN - simple traversal of user datagram protocol (UDP) through network address translators (NATs)," RFC 3489, March 2003.

[14] S. Saroiu, P. Gummadi, S. Gribble *et al.*, "A measurement study of peer-to-peer file sharing systems," University of Washington, Tech. Rep., 2001.

[15] (2008) The Apt-Torrent website. [Online]. Available: http://sianka.free.fr/

[16] (2008) The DebTorrent website. [Online]. Available: http://debtorrent.alioth.debian.org/

[17] M. J. Freedman, E. Freudenthal, and D. Mazières, "Democratizing content publication with Coral," in *NSDI*. USENIX, 2004, pp. 239–252.

[18] G. Pierre and M. van Steen, "Globule: a collaborative content delivery network," *IEEE Communications Magazine*, vol. 44, no. 8, pp. 127–133, 2006.

[19] P. Shah, J.-F. Pâris, J. Morgan, J. Schettino, and C. Venkatraman, "A P2P based architecture for secure software delivery using volunteer assistance," in *8th International Conference on Peer-to-Peer Computing 2008 (P2P'08)*.

[20] S. Annapureddy, M. J. Freedman, and D. Mazières, "Shark: Scaling file servers via cooperative caching," in *NSDI*. USENIX, 2005.