RARAR: Random Access to RAR Archives

Elias Benbourenane
School of Computing and Information
University of Pittsburgh
Pittsburgh, USA
elb222@pitt.edu

***Abstract*—This paper presents RARAR (Random Access RAR)[1], a Python library that enables efficient access to RAR archives without requiring complete downloads. Our implementation supports multiple source types (HTTP URLs, local files, and file-like objects) while achieving significant bandwidth reduction for remote archive access. We discuss the technical challenges of parsing RAR formats, detail our architecture, and demonstrate bandwidth savings exceeding 99% for file listing operations compared to traditional approaches.. While currently limited to uncompressed files, our framework provides a foundation for future development in random-access archive handling.**

***Index Terms*—random access, RAR archives, Python library, HTTP Range requests, bandwidth optimization, file extraction**

I. Introduction

File archiving remains a fundamental practice for efficiently distributing file collections, particularly in cloud environments where bandwidth and storage constraints are critical considerations. The RAR format, despite its proprietary nature, continues to see widespread adoption due to its robust compression capabilities and widespread tool support. However, traditional RAR handling approaches require downloading entire archives before accessing individual files—a significant limitation when working with large remote archives.

Random access to archive contents would provide substantial benefits across numerous applications:

- Previewing archive contents without full downloads
- Extracting specific files from large archives
- Streaming media content stored within archives
- Enabling search engine indexing of archive contents

Unlike formats like ZIP that provide a central directory of contents at the file's end [1], the RAR format poses additional challenges. RAR archives do not contain a single index of all files; instead, file metadata (headers) are interspersed throughout the archive [2]. This makes remote selective access non-trivial, as the archive must be parsed sequentially to locate desired files.

RARAR addresses this limitation by providing a Python package that implements random access to RAR archives. While supporting local files, in-memory buffers, and other file-like objects, the library emphasizes HTTP sources due to their significant practical benefits. Our implementation leverages HTTP Range requests per RFC 7233 [3] to selectively retrieve file segments, dramatically reducing bandwidth requirements and improving access efficiency.

The RARAR library is available as open-source software on GitHub: https://github.com/eliasbenb/RARAR.py. This paper presents the design, implementation, and performance characteristics of the library.

A. Problem Statement

Current approaches to accessing remote archives exhibit substantial inefficiencies, particularly as archive sizes increase. Consider extracting a 100KB document from a 100GB remote archive—the conventional workflow requires:

1) Downloading the entire 100GB archive
2) Extracting the 100KB target file
3) Discarding 99.9999% of the downloaded data

This approach suffers from several critical issues:

- Excessive bandwidth consumption and associated costs
- Storage limitations on client devices
- Time delays during the download process
- Environmental impact due to unnecessary data transfer

For instance, Damien Mannion illustrates in a blog post the challenges of random access with ZIP files. He describes a scenario where downloading an entire 250GB archive, just to extract a few specific files, would take approximately 60 minutes [1]. This scenario motivates techniques for random access to archives, so that specific files can be extracted without transferring the entire archive.

B. Contribution

This paper makes the following contributions:

- A comprehensive analysis of the RAR file format from a random-access perspective
- Development of a Python library enabling random access to RAR archives via multiple source types
- Implementation of an efficient RAR parsing algorithm
- Empirical evaluation of bandwidth-saving potential in HTTP-based RAR access with HTTP Range requests

II. Related Work

A. Random Access in Other Archive Formats

Random access in RAR archives is a novel and unexplored area. Existing research on random access has primarily focused on formats other than RAR, such as:

[1] Available at: https://github.com/eliasbenb/RARAR.py

- **ZIP**: The ZIP format inherently supports efficient random access through its central directory structure located at the file's end. Tools and scripts have been devised to exploit this for partial download: one can fetch the central directory via a byte range request, then know the offset of the desired file and fetch only that part [1], [4].
- **LZOP**: The LZOP format enables random access to compressed data blocks, as demonstrated by the lzopfs project [5], allowing efficient extraction of specific files.

B. HTTP Range Requests in Content Delivery

HTTP Range requests have proven valuable in various content delivery scenarios such as video streaming and resumable downloads. These applications demonstrate how requesting specific byte ranges optimizes data transfer and reduces bandwidth waste.

Other systems like httpfs+ (2022) essentially treat a remote HTTP directory as a filesystem by using partial fetches to simulate file reads [6]. Our work extends these techniques to RAR archive access. To our knowledge, RARAR represents the first implementation of HTTP-based random access for RAR archives.

III. Technical Background

A. RAR File Format

The RAR archive format, developed by Eugene Roshal and maintained by win.rar GmbH, uses a proprietary structure that has been partially reverse-engineered by the community. RAR is essentially a linear sequence of blocks, each with a header and associated data [2], [7], [8].

1) Marker block (RAR signature)
2) Archive header block (MAIN_HEAD)
3) File header blocks (FILE_HEAD, one per file)
4) End of archive block

In other words, a typical ”RAR file has a MAIN_HEAD structure followed by multiple FILE_HEAD structures” [8].

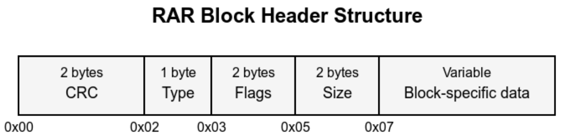

Fig. 1. RAR Block Header Structure

1) File Header Structure: File headers are crucial for random access as they contain location and extraction information. The header extends the basic block structure with additional fields (Table I), including file size, packed size, attributes, and filename—all essential for selective file access.

B. Random Access Challenges

Given this structure, random access in RAR is challenging. Without a central index like in ZIP archives, the only way to discover files and their locations is to parse the headers in sequence. When streaming a RAR archive you initially won't have any information about any of the files-no names, sizes, or file counts/locations until you read through it.

TABLE I
RAR File Header Fields (after the basic header)

Offset	Size	Description
0x00	4 bytes	Packed size (low part)
0x04	4 bytes	Unpacked size (low part)
0x08	1 byte	Host OS (0=MS DOS, 1=OS/2, etc.)
0x09	4 bytes	File CRC32
0x0D	4 bytes	File time (DOS format)
0x11	1 byte	Unpacker version
0x12	1 byte	Method (0x30=Store, 0x31=Fastest, etc.)
0x13	2 bytes	Filename size
0x15	4 bytes	File attributes
0x19	Variable	Optional high pack/unpack sizes
	Variable	Filename (length from offset 0x13)

Moreover, RAR supports an optional solid archive mode, where multiple files are compressed together as one continuous data stream. In a solid archive, even if one knows the byte range of a file's data, that file cannot be independently decompressed without first processing the preceding data in the solid block [2], [7].

C. HTTP Range Requests

HTTP Range requests enable clients to request specific portions of resources, a capability essential for our implementation when working with remote archives. The protocol, defined in RFC 7233 [3], uses the Range header to specify byte ranges and responds with 206 Partial Content status.

1) Range Request Specification: The HTTP/1.1 protocol defines the Range header field in RFC 7233. A typical Range request header has the format:

```
Range: bytes=start-end
```

Where `start` and `end` are byte positions. For example:

```
Range: bytes=0-499 // First 500 bytes
Range: bytes=500-999 // Second 500 bytes
Range: bytes=-500 // Last 500 bytes
Range: bytes=500- // All bytes from position 500
```

2) Server Support Considerations: Range request support varies across servers. Our implementation detects support through:

- Confirming 206 or 200 response codes for range requests
- Checking for the `Accept-Ranges` header
- Validating requested and returned content lengths

3) RAR Format Variations: RAR has evolved through versions 1.3, 2.0, 3.0, 4.0, and 5.0, each introducing changes to header structures and compression algorithms. Our implementation targets the widely-adopted RAR 3.0 and 4.0 formats.

4) Reverse Engineering Approach: Without official documentation, our understanding derives from:

- Analysis of open-source implementations
- Community-documented specifications
- Testing against sample archives

IV. Design and Implementation

A. Architecture Overview

RARAR is designed around the principle of minimal data transfer, downloading only the bytes necessary to accomplish the required task. The library consists of three main components:

1) **Source Handlers**: Abstractions for file-like objects, including HTTP sources, local files, and memory buffers
2) **Version Handlers**: Factory pattern implementation for automatic RAR version detection
3) **File Extractor**: Core logic for parsing headers and extracting files

1) Source Handler Abstraction: Our adapter pattern enables archive access regardless of source type, maintaining separation between parsing logic and data access methods. Compatible sources must implement standard Python `io` module interfaces [9]

We implemented a custom HttpFile class that mimics Python's built-in file object, but under the hood it performs HTTP range requests to fulfill read and seek operations.

Required methods include:

- `seek(position)`: Navigate to specific byte positions
- `read(size)`: Read specified byte counts from current position

B. RAR Parsing Algorithms

RARAR's core innovation lies in iterating through archives without downloading compressed content. The implementation parses only necessary headers while skipping file data, achieving substantial bandwidth reduction.

Algorithm 1 demonstrates our efficient header parsing approach. We initially read a 128-byte chunk containing most headers, performing additional reads only when necessary. This strategy minimizes network requests, particularly valuable for HTTP sources.

One optimization in our implementation is to minimize the number of HTTP requests during this scanning phase. Instead of fetching each header with a separate tiny request, RARAR reads in moderate-sized chunks (e.g. 256 KB) that likely contain multiple file headers at once.

V. Performance Evaluation

A. Experimental Setup

We evaluated RARAR using diverse archives under controlled conditions:

- **Client**: Python 3.13 on Ubuntu 24.04
- **Server**: RClone WebDAV server on Ubuntu 24.04
- **Network**: 1Gbps LAN with simulated latency using clumsy
- **Test Archives**: RAR archives from 10MB to 50GB containing 10-250 files

Algorithm 1: Parsing File Headers

```
Input: position in the file
Output: file info object and next position
$headerChunk \leftarrow ReadBytes(position, 128)$
$headType, headFlags, headSize \leftarrow ParseHeader(headerChunk[0:7])$
if $headType \neq RAR3\_BLOCK\_FILE$ then
    return $null, position + headSize$
if $headSize > len(headerChunk)$ then
    $headerChunk \leftarrow headerChunk + ReadBytes(position + 128, headSize - 128)$
$packSize, unpSize, method, fileCRC, nameSize \leftarrow ParseFileFields(headerChunk[7:])$
if $headFlags \; \& \; FLAG\_HAS\_HIGH\_SIZE$ then
    $highPackSize, highUnpSize \leftarrow ParseHighSizes(headerChunk)$
    $packSize \leftarrow packSize + (highPackSize \ll 32)$
    $unpSize \leftarrow unpSize + (highUnpSize \ll 32)$
$fileName \leftarrow ParseFileName(headerChunk, nameSize, headFlags)$

$isDirectory \leftarrow (headFlags \; \& \; FLAG\_DIRECTORY) = FLAG\_DIRECTORY$
$dataOffset \leftarrow position + headSize$
$nextPos \leftarrow dataOffset$
if $headFlags \; \& \; FLAG\_HAS\_DATA$ then
    $nextPos \leftarrow nextPos + packSize$
$fileInfo \leftarrow CreateFileInfo(fileName, unpSize, ...)$
return $fileInfo, nextPos$
```

B. Bandwidth Usage

RARAR demonstrates exceptional bandwidth efficiency during archive listing operations. Table II shows bandwidth usage for listing files across various archive sizes, where only file headers are retrieved rather than full archive contents.

The key finding is that bandwidth usage remains constant regardless of archive size, scaling linearly with file count at approximately 32KB per file due to our header-only parsing approach.

If we denote the archive size as S and the number of files as N, the bandwidth savings can be approximated with:

$$\text{Bandwidth Savings} \approx \frac{S - (N \times 32KB)}{S} \times 100\% \qquad (1)$$

This formula accurately models our experimental results. For instance, with a 50GB archive containing 250 files, RARAR transferred only 7.82MB of data, representing a 99.980% reduction from conventional download methods. This measured result aligns with the formula's prediction of 99.984% savings. Similarly, the worst-performing case (250 files in a 10MB archive), also accurately aligns with the formula's prediction of 21.875% savings.

These bandwidth savings apply specifically to the file listing operation. File extraction requires additional bandwidth proportional to the target file size, though RARAR still provides advantages by transferring only the requested file data rather than the entire archive. For example, extracting a 10MB file from a 1GB archive requires only 10MB of transfer (plus the minimal header overhead outlined in Table II) versus 1GB with traditional methods. In edge cases where users extract all files or the entire archive consists of a single file, RARAR's bandwidth consumption exactly matches that of traditional methods.

TABLE II
BANDWIDTH COMPARISON FOR FILE LISTING: RARAR VS. TRADITIONAL METHODS

File Count	Archive Size	Bandwidth	Bandwidth Savings
10	10MB	0.32MB	96.800%
10	100MB	0.32MB	99.680%
10	1GB	0.32MB	99.970%
10	10GB	0.32MB	99.997%
10	50GB	0.32MB	99.999%
50	10MB	1.57MB	84.300%
50	100MB	1.57MB	98.430%
50	1GB	1.57MB	99.850%
50	10GB	1.57MB	99.980%
50	50GB	1.57MB	99.997%
250	10MB	7.82MB	21.800%
250	100MB	7.82MB	92.180%
250	1GB	7.82MB	99.240%
250	10GB	7.82MB	99.920%
250	50GB	7.82MB	99.980%

C. Operation Latency

Our experiments focused on the latency of archive listing operations, which depend primarily on network conditions due to multiple small HTTP requests for reading headers. Table III shows listing performance under various network latencies.

$$T = n \times (L + P) \tag{2}$$

Where T is total time, n is the number of files, L is network latency per request, and P is processing time per header (both by the client and server). At 1ms latency, listing time increases from 46ms for 10 files to 411ms for 250 files (approximately 40ms per additional 10 files). At 100ms latency, the same operation ranges from 2,374ms to 33,961ms, showing a much steeper increase (approximately 1,317ms per additional 10 files). This linear scaling confirms that archive listing time is dominated by network round-trip times rather than processing overhead.

Given the impact latency has on operation time, RARAR implements several optimizations to reduce it:

- Batch reading of adjacent headers
- Predictive prefetching for likely-needed blocks
- Dynamic chunk sizing based on network conditions

TABLE III
NETWORK CONDITION IMPACT ON RARAR FILE LISTING PERFORMANCE

Latency	Number of Files	Time to List
1ms	10	46ms
1ms	50	102ms
1ms	250	411ms
50ms	10	1,587ms
50ms	50	5,089ms
50ms	250	22,636ms
100ms	10	2,374ms
100ms	50	7,639ms
100ms	250	33,961ms

The other critical operation for RARAR is file extraction. For example, to retrieve a 5 MB file at the end of a 500 MB archive over a 20 Mbit/s connection with 50ms round-trip time, a full download would take approximately $\frac{500\text{ MB}\times 8\text{ Mbit/MB}}{20\text{ Mbit/s}} =$ 200 seconds. In contrast, RARAR would require several range requests to parse headers (1-2 requests at 50ms each, totaling 50-100ms) followed by a single request to exclusively download the 5 MB file ($\frac{5\text{ MB}\times 8\text{ Mbit/MB}}{20\text{ Mbit/s}} = 2$ seconds). The total extraction time would be approximately 2.05-2.10 seconds—a 99% reduction in time compared to the traditional approach.

As with the bandwidth experiment, these latency measurements reflect file listing operations only, where RARAR must parse headers sequentially. File extraction latency includes this initial listing overhead plus the time to transfer the target file data. For example, extracting a 5MB file from a 500MB archive at the end position requires: (1) header parsing time (shown in Table III), and (2) transfer time for the 5MB file, which is dependent on network bandwidth.

It it also worth noting that the physical position of target files for extraction within an archive significantly impacts completion time. Files at the beginning of an archive require parsing only the initial headers, while files at the end require sequential parsing of all preceding headers. This positional dependency introduces variable latency, with worst-case scenarios occurring when extracting files located near the archive's end. Our prior example assumed this worst-case scenario.

D. Scalability Analysis

Our analysis demonstrates excellent scalability characteristics. Bandwidth usage remains constant regardless of archive size (Table II), while operation latency scales linearly with file count (Table III). This enables efficient handling of arbitrarily large archives containing numerous files.

VI. LIMITATIONS AND FUTURE WORK

A. Current Limitations

Despite its advantages, RARAR currently has several limitations:

1) **Compression Support**: Only uncompressed ("Store") files are supported. Files using RAR's compression algorithms cannot be extracted without downloading compressed blocks and implementing decompression.

2) **Server Requirements**: Remote servers must support HTTP Range requests for RARAR to function correctly.
3) **Multi-Part Archives**: Archives split across multiple files are not yet supported.
4) **Encrypted Archives**: Password-protected archives cannot be processed due to the need to decrypt the entire archive structure.

B. Future Work

Several promising extensions exist:

1) Compressed File Support: Implementing support for common RAR compression methods would significantly enhance utility. This requires:

- Integration of RAR decompression algorithms
- Analysis of partial decompression feasibility
- Optimization of compressed data retrieval

A promising approach involves developing partial decompression strategies that identify and download only necessary compressed blocks before performing targeted decompression.

2) RAR5 Support: Supporting RAR5 would extend versatility by accommodating the latest format version. Our extensible factory pattern facilitates adding new version handlers while maintaining compatibility with existing formats.

C. Applications and Use Cases

The random access capabilities introduced by RARAR enable numerous practical applications across diverse domains:

1) File Server Integration: The original inspiration for RARAR was to integrate on-the-fly RAR archive access into a WebDAV server that mirrors an external repository. This integration would allow users to access specific files within RAR archives without the user or the mirror server needing to download the entire file.

2) Cloud Storage Systems: Cloud storage providers could integrate RARAR to enhance user experience when accessing archived content. Instead of requiring users to download complete archives, providers could offer direct file previews and selective extraction from RAR archives.

Google Drive already supports this feature for ZIP files, allowing users to preview and extract files without downloading the entire archive. RARAR could extend similar functionality to RAR archives.

3) Content Distribution Networks: CDNs could leverage RARAR to optimize delivery of archived content. Rather than storing and serving individual files, CDNs could maintain single compressed archives while allowing clients to request specific components. This approach reduces storage overhead while maintaining efficient content delivery, particularly beneficial for distributing large software packages, game assets, or multimedia collections.

4) Media Streaming Services: Streaming platforms could utilize RARAR to manage archived media content efficiently. Large collections of video or audio files could remain in compressed archives while allowing real-time extraction and streaming of specific content on demand, reducing storage costs.

5) Backup and Disaster Recovery: Backup systems could implement RARAR to enable granular file restoration. Instead of restoring complete archive volumes, administrators could selectively recover specific files or directories.

VII. Conclusion

RARAR represents a novel approach to working with RAR archives. By combining careful format parsing with HTTP Range requests, our solution enables efficient file listing and extraction without full archive downloads.

The demonstrated bandwidth savings when listing contents of a 50GB archive while transferring only 0.32MB highlight significant efficiency improvements. These savings extend to file extraction, where only the required file data needs to be transferred.

This work provides both practical tools for developers and demonstrates techniques applicable to other archive formats, advancing the state of remote file access optimization.

References

[1] D. Mannion, ”Extracting Files from a Remote Zip Archive,” djmannion.net, 2022. [Online]. Available: https://www.djmannion.net/partial_zip/index.html

[2] ”RAR Version 4.20 - Technical Information,” rescene.wikidot.com, 2014. [Online]. Available: http://rescene.wikidot.com/rar-420-technote

[3] R. Fielding et al., ”Hypertext Transfer Protocol (HTTP/1.1): Range Requests,” RFC 7233, Internet Engineering Task Force, Jun. 2014.

[4] GDAL Contributors, ”GDAL Virtual File Systems: /vsimem, /vsizip, /vsitar, /vsicurl,” GDAL, 1998–2025. [Online]. Available: https://gdal.org/en/stable/user/virtual_file_systems.html

[5] D. Vasilevsky, ”lzopfs: A FUSE Filesystem for Transparent Random Access to Compressed LZOP Archives,” GitHub, 2011. [Online]. Available: https://github.com/vasi/lzopfs

[6] D. Guillen Fandos, ”httpfs+: A FUSE Based Filesystem over HTTP,” davidgf.net, 2022. [Online]. Available: https://www.davidgf.net/2022/10/01/httpfs/index.html

[7] E. Roshal, ”RAR Archive File Format,” Technical Specification, RARLAB, 2019.

[8] S. Jeff, ”Documentation of the RAR Format,” BitJS Documentation. [Online]. Available: https://codedread.github.io/bitjs/docs/unrar.html

[9] Python Software Foundation, ”io — Core Tools for Working with Streams,” Python Documentation, version 3.13, 2024.

[10] M. Constantinescu, A. Rusu, and I. Bucur, ”Random Extraction from Compressed Data: A Practical Study,” IBM Research, 2013.

[11] E. Benbourenane, ”RARAR.py: Random Access RAR,” GitHub Repository, 2025. [Online]. Available: https://github.com/eliasbenb/RARAR.py