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% 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

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 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.

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

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

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. Existing RAR Handling Libraries

Several libraries provide RAR archive handling capabilities, though none are designed for random access:

- **UnRAR**: The official C library from RARLAB provides extraction functionality but requires local access to complete archives.
- **Python-rarfile**: A Python module wrapping the UnRAR library or command-line tools. While feature-complete, it requires full archive availability before extraction.

B. Random Access in Other Archive Formats

Research on random access has primarily focused on formats other than RAR:

- **ZIP**: The ZIP format inherently supports efficient random access through its central directory structure located at the file's end. Libraries like Python's `zipfile` can parse this directory without downloading the entire archive, though full file access remains necessary for extraction.

- **LZOP**: The LZOP format enables random access to compressed data blocks, as demonstrated by the lzopfs project [5], allowing efficient extraction of specific files. However, LZOP lacks the widespread adoption of RAR or ZIP.

C. 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.

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. The format consists of sequential blocks:

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

Each block follows a consistent structure (Figure 1) with a header containing block type, size, and various flags.

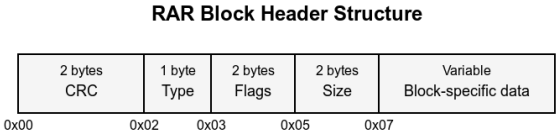

Fig. 1. RAR Block Header Structure

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.

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)

B. 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 principle, downloading only the bytes necessary to accomplish specific tasks. 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.

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.

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 $fileInfo, nextPos$

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

B. Bandwidth Usage

RARAR demonstrates exceptional bandwidth efficiency. Table II shows bandwidth savings across various archive sizes. 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.

For a 50GB archive with 250 files, RARAR required only 7.82MB of data transfer—a 99.98% reduction compared to traditional methods.

TABLE II
Bandwidth Comparison: 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

Operation latency depends primarily on network conditions due to multiple small HTTP requests. Table III shows performance under various network latencies. Listing 250 files takes 411ms at 1ms latency but increases to 33,961ms at 100ms latency.

To mitigate latency impact, we implemented:

- 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 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

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.

3) Server Integration: Integrating RARAR into web servers could enable efficient file serving through:

- Server-side random access capabilities
- Client-specific file range requests
- On-the-fly decompression and extraction

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 accessing a 50GB archive while transferring only 0.32MB highlight significant efficiency improvements. Although currently limited to uncompressed files, RARAR establishes a foundation for comprehensive random-access archive handling.

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] E. Benbourenane, "RARAR.py: Random Access RAR," GitHub repository, Available: https://github.com/eliasbenb/RARAR.py, 2025.
[2] E. Roshal, "RAR archive file format," Technical Specification, RARLAB, 2019.
[3] R. Fielding et al., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests," RFC 7233, Internet Engineering Task Force, June 2014.
[4] Python Software Foundation, "io — Core tools for working with streams," Python Documentation, version 3.13, 2024.
[5] D. Vasilevsky, "lzopfs: A FUSE filesystem for transparent random access to compressed LZOP archives," GitHub repository, Available: https://github.com/vasi/lzopfs, 2011.