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, file-like objects) and achieves significant bandwidth reduction for remote archive access. We discuss the technical challenges of working with RAR formats, detail our architecture, and demonstrate bandwidth savings of over 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 common practice for distributing collections of files efficiently, especially on the cloud where bandwidth and storage are limited. The RAR format, despite being proprietary, continues to be widely used due to its popularity and simplicity. However, traditional approaches to working with RAR archives require downloading the entire archive before any file within it can be accessed or extracted. This becomes particularly problematic when dealing with large archives hosted remotely.

The ability to randomly access specific portions of an archive without downloading the entire file would be valuable in numerous scenarios, such as:

- Previewing archive contents before deciding to download
- Extracting a single small file from a large archive
- Streaming media files stored within archives
- Indexing archive contents for search engines

RARAR addresses this gap by providing a Python package that enables random access to RAR archives. While it supports local files, in-memory buffers, and other file-like objects, the library primarily focuses on HTTP sources due to the massive benefits it provides for remote archives. The implementation leverages HTTP Range requests to efficiently retrieve specific file segments, thereby reducing bandwidth consumption and improving overall performance.

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

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

A. Problem Statement

Traditional methods for accessing remote archives suffer from significant inefficiencies, especially as sizes grow. Consider extracting a 100KB document from a 100GB remote archive - conventional approaches require:

1) Downloading the entire 100GB archive
2) Extracting the 100KB file locally
3) Discarding 99.9999% of the unused archive

This approach is not only inefficient but often impractical due to:

- Bandwidth constraints and costs
- Storage limitations on client devices
- Time delays during the download process
- Environmental impact of unnecessary data transfer

B. Contribution

The primary contributions of this paper are:

- A detailed analysis of the RAR file format from a random access perspective
- Development of a Python library that enables random access to RAR archives via HTTP and other sources
- Implementation of a RAR parsing algorithm
- Empirical evaluation of the bandwidth savings achieved through this approach

II. Related Work

A. Existing RAR Handling Libraries

Several libraries exist for handling RAR archives in various programming languages. However, none of them are designed with random access as a feature:

- **UnRAR**: The official C library from RARLAB provides extraction capabilities but requires local access to the full archive.
- **Python-rarfile**: A Python module for RAR archive reading that wraps the UnRAR library or command-line tools. While feature-rich, it still requires downloading the entire archive before extraction.
- **lzopfs**: A FUSE-based filesystem for parsing LZOP archives with random access.

B. Random Access in Other Archive Formats

While random access to RAR archives has not been well-explored, some research has been conducted on random access to other archive formats:

- **ZIP**: The ZIP format inherently supports better random access due to its directory structure being located at the end of the file. Libraries like `zipfile` in Python can read the central directory without downloading the entire archive, though they still require full file access for extraction.
- **LZOP**: The LZOP format allows for random access to compressed data blocks, as demonstrated by the lzopfs project, enabling efficient extraction of specific files. However, it is not as widely used as RAR or ZIP.

C. HTTP Range Requests in Content Delivery

HTTP Range requests have been utilized in various content delivery applications like video streaming and resuming downloads. These applications allow clients to request specific byte ranges of a file, making data transfer much more efficient and less wasteful.

Our work builds upon these foundations, applying range request techniques specifically to the challenges of RAR archive access. To our knowledge, RARAR represents the first implementation of random access to RAR archives over HTTP.

III. Technical Background

A. RAR File Format

The RAR archive format was developed by Eugene Roshal and is maintained by win.rar GmbH. While the format is proprietary, reverse-engineered specifications are available from various sources. The format consists of a series of blocks, including:

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

The RAR format uses a consistent block structure throughout the archive. Following the basic layout outlined in Table I, each block begins with a header that contains information about the block type, size, and various flags.

TABLE I
RAR Block Header Structure

Offset	Size	Description
0x00	2 bytes	CRC of header data
0x02	1 byte	Block type
0x03	2 bytes	Block flags
0x05	2 bytes	Block size
0x07	Variable	Block-specific data

1) File Header Structure: The file header is particularly important for random access as it contains information on where to locate and extract specific files. The file header extends the basic block structure with additional fields, as shown in Table II.

Most notably, the file header includes fields for file size, packed size, file attributes, and file name, all of which are crucial for random access.

TABLE II
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 allow clients to request only a portion of a resource from a server. This capability is essential for our integration with remote files served through HTTP, as it enables requesting only the necessary parts of the RAR archive. The Range header in HTTP requests specifies which bytes of the resource to return, and servers respond with a 206 Partial Content status containing the desired file range.

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: Not all HTTP servers support Range requests. Our implementation detects whether a server supports this feature through a combination of:

- Confirming a successful 206 or 200 response code when requesting a range.
- Finding an `Accept-Ranges` header in the response.
- Comparing the requested and returned content lengths.

3) RAR Format Variations: The RAR format has evolved through several versions (RAR 1.3, 2.0, 3.0, 4.0, and 5.0), each with subtle differences in header structures and compression algorithms. Our implementation focuses primarily on the most widely-used RAR 3.0 and 4.0 formats.

4) Reverse Engineering Approach: Without official documentation, our understanding of the RAR format was derived from:

- Analysis of existing open-source implementations
- Community-documented specifications
- Testing against archive samples

IV. Design and Implementation

A. Architecture Overview

RARAR is designed around the principle of minimal data transfer, 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**: A factory design pattern for automatic RAR version detection and handling
3) **File Extractor**: Logic for parsing RAR headers and extracting individual files from archives

1) Source Handler Abstraction: RARAR uses an adapter design pattern for accessing archive data regardless of the source. This abstraction allows the core parsing logic to remain independent of the archive's source (local file, HTTP URL, memory buffer, etc.).

Any source compatible with the `io` module in Python can be used as a source handler. The library provides built-in handlers for HTTP URLs and local files.

Specifically, the source handler must implement two methods for reading data:

- `seek(position)`: Move to a specific byte position in the source
- `read(size)`: Read a specific number of bytes from the current position

B. RAR Parsing Algorithms

The core of RARAR is its ability to iterate through files in an archive without downloading the compressed content. The implementation parses only the necessary headers and file information, ignoring the compressed data to vastly reduce bandwidth usage.

Parsing file headers requires careful handling of variable-length fields and optional components. Algorithm 1 illustrates how we extract the necessary information while minimizing the number of reads.

The parser initially reads a 128-byte chunk that typically contains the entire header, avoiding multiple small reads. If the header exceeds this initial chunk (which is rare), we perform a single additional read for the remaining data. Variable-length fields like filenames are handled efficiently by first reading the fixed-size metadata that specifies their length, then extracting the exact bytes needed.

Optional components like high pack/unpack sizes are only read when the header flags indicate their presence. The parser uses a BytesIO stream to sequentially extract fields from the pre-read buffer rather than making separate read calls for each field. This buffered approach means that instead of making 10-15 separate reads for individual header fields, we typically need only one or two reads total, significantly reducing overhead, especially for remote sources using HTTP Range requests.

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
    $nextPos \leftarrow nextPos + packSize$
$fileInfo \leftarrow$
  $CreateFileInfo(fileName, unpSize, packSize,$
                  $method, fileCRC, isDirectory, dataOffset, nex$

return $fileInfo, nextPos$
```

V. Performance Evaluation

A. Experimental Setup

To evaluate RARAR's performance, we conducted a series of experiments using archives of various sizes and compositions. The testing environment consisted of:

- **Client**: Python 3.13 on Ubuntu 24.04
- **Server**: RClone WebDav server on Ubuntu 24.04
- **Network**: 1Gbps LAN with simulated latency and bandwidth constraints using clumsy
- **Test Archives**: Multiple RAR archives ranging from 10MB to 50GB with varying numbers of files (10 to 250)

B. Bandwidth Usage

The primary advantage of RARAR is its minimal bandwidth requirements. We evaluated the library's performance by testing against archives of various sizes, measuring the amount of data transferred to perform common operations. The results demonstrate exceptional bandwidth efficiency and scalability.

Table III summarizes the bandwidth savings achieved by RARAR for different archive sizes. The results demonstrate how RARAR can be used to efficiently access large archives. Notably, bandwidth usage is independent of archive size and instead scales linearly with the number of files in the archive, with each file requiring approximately 32KB of data transfer. This is due to RARAR's approach of parsing only file headers while ignoring data blocks.

For a 50GB archive containing 250 files, RARAR required only 7.82MB of data transfer to list all files in the archive. This represents a 99.98% reduction in bandwidth usage compared to traditional methods requiring full archive download.

TABLE III
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

The operation latency of RARAR is influenced primarily by network conditions, particularly latency, due to the multiple small HTTP requests required for parsing archive structures. Our experiments measured the time taken to list files in RAR archives under different network latencies.

To simulate various network conditions, we used the clumsy tool to introduce artificial latency. The tests were conducted against a 100MB archive.

Table IV shows the impact of varying network latencies on the time required to list the contents of RAR archives with different numbers of files. As expected, higher network latency results in increased operation time. For instance, listing the contents of an archive with 250 files takes approximately 411 ms with a network latency of 1 ms, but this increases to 33,961 ms with a latency of 100 ms.

Because of the large impact network latency has on RARAR's performance with HTTP sources, several optimizations have already been implemented:

- Batch reading of headers where possible
- Predictive prefetching of likely-to-be-needed blocks
- Adaptive chunk sizing based on network conditions

D. Scalability Analysis

As demonstrated in the results of Table III, RARAR's bandwidth usage has perfect scalability with archive size, and, as demonstrated in Table IV, the latency of operations scales linearly with the number of files in the archive. This means that RARAR can perfectly handle archives of any size and efficiently handle large numbers of files.

TABLE IV
NETWORK CONDITION IMPACT ON RARAR PERFORMANCE

Latency	Number of Files	Time to List
1ms	10	46ms
1ms	50	102ms
1ms	250	411ms
50ms	10	1587ms
50ms	50	5089ms
50ms	250	22636ms
100ms	10	2374ms
100ms	50	7639ms
100ms	250	33961ms

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 avenues for extending RARAR's capabilities include:

1) Compressed File Support: Implementing support for the most common RAR compression methods would greatly expand the library's utility. This would require:

- Implementing or integrating RAR decompression algorithms
- Analyzing whether partial decompression from arbitrary positions is feasible
- Optimizing to minimize the amount of compressed data needed

One promising approach would be to develop a partial decompression strategy that identifies blocks containing the target file and downloads only the necessary compressed blocks. From there, the library could perform targeted decompression of those blocks, allowing for random access to compressed files.

2) RAR5 Support: RAR5 is the latest version of the RAR format, introducing new features and compression methods. Implementing RAR5 support would expand RARAR's versatility and would require a rework of the header parsing logic to accommodate the new block structures and compression methods.

The current implementation of RARAR uses a factory design pattery, designed to be extensible, allowing for the addition of new version handlers as needed. This would enable RARAR to support both RAR3 and RAR5 formats seamlessly.

3) Integration into Web Servers: Integrating RARAR with web-based file servers was our original inspiration for the project. It could enable servers to much more efficiently serve files. We envision a system where:

- A web server can serve RAR archives with random access capabilities
- Clients can request specific files or ranges from the archive
- The server handles decompression and file extraction on-the-fly

VII. Conclusion

RARAR.py represents a novel approach to working with RAR archives, enabling random access with minimal bandwidth requirements. By carefully parsing RAR file structures and leveraging HTTP Range requests, our solution enables efficient listing and extraction of files from remote archives without downloading entire archives.

The demonstrated bandwidth savings in accessing a 50GB archive while transferring only 0.32MB of data highlights the significant efficiency improvements possible with this approach. While current limitations restrict the library to uncompressed files, the foundation established by RARAR opens pathways for future work on more comprehensive random access archive handling.

This project contributes both a practical tool for developers working with remote archives and demonstrates techniques that could be applied to other archive formats to enable more efficient remote file access.

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.