Developing ZIP Support in the XRootD Client

August 2020

A CERN summer project carried out remotely during the COVID-19 pandemic.

Author:
Sonia M Marshall

Supervisor:
Michał Simon

With support from:
Elvin Alin Sindrilaru

Department:
IT-ST-PDS

Developing ZIP Support in the XRootD Client

Contents

Abstract

A ZipArchive class has been developed for the XRootD client C++ API to provide the functionality to append files to existing ZIP archives and to create new ZIP archives. This goes hand in hand with the existing functionality of extracting individual files from ZIP archives, and has been developed to support the ZIP64 format to enable manipulation of large files, for use by the high energy physics community.

1 Introduction

XRootD is a software framework for fast, low latency and scalable data access that is widely used by the high energy physics community. The XRootD project is an open source collaboration between CERN, SLAC, Duke University, JINR and UCSD. Over the course of 8 weeks, my project has involved developing the ZIP support in the XRootD client - there was already the functionality of extracting a file from a ZIP archive, and I developed the opposite functionality: appending files to ZIP archives. This also required running an XRootD server to be able to test my code, although I was not developing the server itself. The client is implemented in C++ and can be used via a C++ API. This API is used by the LHC experiments frameworks as well as software defined storage solutions such as EOS, the main storage system used at CERN. Additionally the client is available through the command line tools `xrdcp` and `xrdfs`, as well as Python bindings.

ROOT is a data analysis framework and file format commonly used in high energy physics. Since the ROOT file format uses ZIP64 for bundling data, it is therefore useful to have tools available for manipulating ZIP archives easily, and this is why I have been developing the ZIP support in the XRootD client. When dealing with such large quantities of data as is the case in high energy physics, this added functionality of appending a file to an existing archive will be much more efficient than unzipping an archive and re-zipping all the files again with the newly added file.

In this report I detail the structure of the ZIP file format, how I implemented the append functionality, how I tested my generated ZIP archives and finally the work that remains to be done continuing on from this project in the future.

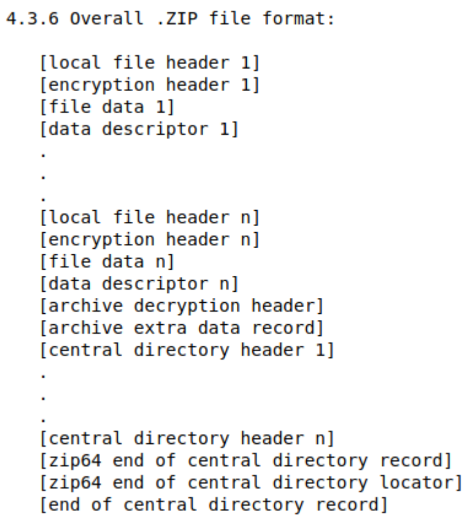
2 ZIP files

The ZIP file format is specified in PKWare's APPNOTE.TXT. Version 6.3 of the specification was used in this project. ZIP archives are used to bundle together multiple files - these files can be compressed, as is a common usage, however this is not required, the files can simply be stored in the archive (uncompressed).

2.1 Headers

A ZIP archive consists of various headers and records containing fields of information, as well as the stored file content (see Figure 1). Each file stored in the archive has a *local file header* (LFH) which contains information about the file. A few of the key fields in the LFH are: filename, compressed and uncompressed size, last modified file time and date and the CRC-32 value which is used for data integrity. The archive can contain many LFHs, each followed immediately by the corresponding *file data*.

At the end of the archive is the central directory. This contains a *central directory file header* (CDFH) for each file, which has all the fields that are in the LFH, plus a few additional fields, including the offset of the corresponding LFH. After the CDFHs there is the *end of central directory record* (EOCD). This contains information about the rest of the central directory, such as the number of entries, the size and the offset of the central directory.



```
4.3.6 Overall .ZIP file format:

   [local file header 1]
   [encryption header 1]
   [file data 1]
   [data descriptor 1]
   .
   .
   .
   [local file header n]
   [encryption header n]
   [file data n]
   [data descriptor n]
   [archive decryption header]
   [archive extra data record]
   [central directory header 1]
   .
   .
   .
   [central directory header n]
   [zip64 end of central directory record]
   [zip64 end of central directory locator]
   [end of central directory record]
```

Figure 1: Structure of a ZIP file (Section 4.3.6 of APPNOTE.TXT).

There are a few other headers and records that can be included in a ZIP file, such as *encryption headers* and *data descriptors*, however these are not relevant to the use case in this project, so I will not go into detail here. Each of the headers has a unique signature which comes before the rest of the fields to identify what type of header follows. For example, the LFH signature is 0x04034b50.

2.2 ZIP64

The ZIP64 format extensions allow the support of very large files. The first of these extensions is the *ZIP64 extended information extra field* in the LFH and CDFH. If one of the fields in the LFH and CDFH is too small to hold the required data, the field is set to -1 (0xFFFF or 0xFFFFFFFF) and the value is stored in the extra field. This only applies to a few specific fields, the most important being the compressed and uncompressed size of the file and the LFH offset. The other extension is the *ZIP64 end of central directory record* (ZIP64 EOCD), accompanied by the *ZIP64 end of central directory locator* (ZIP64 EOCDL). If one of the fields in the EOCD is too small to hold required data, the field is set to -1, the ZIP64 EOCD and ZIP64 EOCDL are created, and the value is stored in the equivalent field in the ZIP64 EOCD. The ZIP64 EOCDL contains the offset of the ZIP64 EOCD. These two records fit in after the CDFHs and before the EOCD.

3 Project Implementation

In my implementation the overall process of how a file is appended to a ZIP archive is as follows:

- the archive is opened and the existing central directory is read and stored
- the local file header for the input file is written over the existing central directory
- the file data is written
- the updated central directory is written
- the archive is closed

It is also possible to create a new ZIP archive, in which case rather than reading and updating an existing central directory, a new central directory is created and written out at the appropriate location. For any reader interested in more detail of the implementation than is provided in this report, the source code can be found in the GitHub repository for the project.

3.1 Milestones

The overall task of adding the full append functionality was broken down into smaller tasks, making it more manageable to progress step by step with the project.

1. Create a simple local ZIP archive from C++
2. Create a large local ZIP archive using ZIP64 format
3. Append files to an existing local ZIP archive
4. Append files to an existing local ZIP archive - ZIP64 format support
5. Use XrdCl synchronous API to create/append to remote ZIP archives
6. Investigate using XrdCl asynchronous API

3.2 ZipArchive API

The following is the final ZipArchive API I have produced in this project:

```
ZipArchive::Open(...)           // open archive, read existing central directory
ZipArchive::Append(...)         // create headers and write local file header to archive
ZipArchive::WriteFileData(...)  // write contents of buffer to archive
ZipArchive::Finalize(...)       // write central directory to archive
ZipArchive::Close(...)          // close archive
```

The functions should be called in the order listed above. WriteFileData() writes a buffer to the archive so will usually be called multiple times in order to write the entire input file data. It is possible to append multiple files to an open archive by calling Append() and WriteFileData() multiple times before finalizing and closing the archive.

3.3 Classes, structs and their members

ZIP archives are encapsulated by the ZipArchive class. This class contains the member functions described above in the API, and additionally some helper functions. A few member variables to note are the following:

```
File                   &archive;       // the ZIP archive
std::vector<CDFH*>      cdRecords;     // stores the new CDFHs for files being appended
EOCD                   *eocd;          // the EOCD
ZIP64_EOCD             *zip64Eocd;     // the ZIP64 EOCD
ZIP64_EOCDL            *zip64Eocdl;    // the ZIP64 EOCDL
std::unique_ptr<char[]> cdBuffer;      // stores existing central directory of an archive
uint64_t                writeOffset;   // stores offset of the archive for writing
```

Each of the headers and records described in Section 2 are encapsulated by a struct, which has a member variable for each field of the header. Note that there is also a struct that represents the *ZIP64 extended information extra field*. The EOCD, ZIP64 EOCD and ZIP64 EOCDL each have two constructors: one for use when reading the central directory of an existing archive, and one used when creating a new archive or adding the ZIP64 format extension. For example, the EOCD constructors:

```
EOCD( const char *buffer ) {...}    // used when reading from existing ZIP archive
EOCD( LFH *lfh, CDFH *cdfh ) {...}  // used when creating new ZIP archive
```

Each of the headers has a Write() method, which concatenates the fields together in the correct order into a buffer and writes this buffer to the archive at the given offset.

```
void Write( File &archive, uint64_t writeOffset ) {...}
```

3.4 Local vs remote files

As can be seen from the milestones shown in Section 3.1, the code initially generated a local ZIP archive. At this stage, Linux syscalls were used for the stat, open, read, write and close operations. Using these, the file is referred to by a file descriptor. In order to generate a remote archive on the XRootD server, these syscalls were replaced by the equivalents from the XRootD client C++ API. Rather than dealing with file descriptors, an XrdCl::File object was created, and the synchronous function calls of open, read, write and close from the XrdCl::File class, and stat from the XrdCl::Filesystem class were used. The XRootD URL of the archive (i.e. root://host//path/to/archive.zip) was used to open the file, rather than the filename which was used in the local version. A comparison of opening a local versus a remote archive can be seen below.

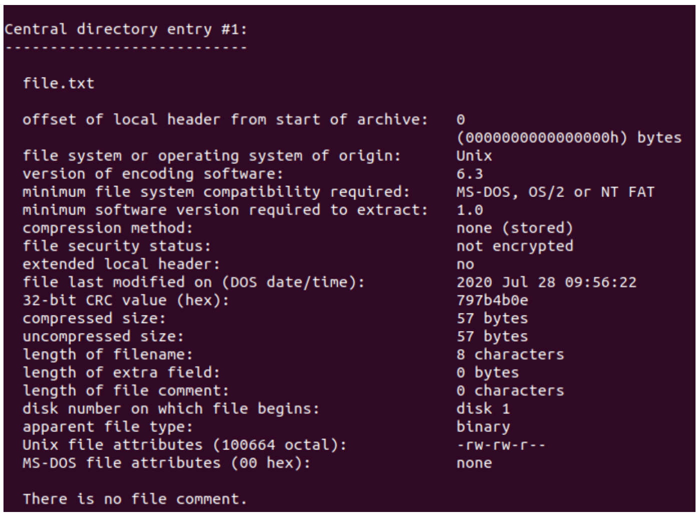
```
int archiveFd = open(   archiveFilename.c_str(),
                        O_RDWR,
                        S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH );
XRootDStatus st = archive.Open( archiveUrl,
                                OpenFlags::Update,
                                Access::UR | Access::UW | Access::GR | Access::OR );
```

4 Testing

In order to test my code, I created an executable that used the ZipArchive API to open a ZIP archive, append a file and then close the archive. I ran this with both small files and a file larger than 4GB, so that I could test both the simple case and the case where the ZIP64 format is required. I tested creating a new ZIP archive, and appending to an existing archive in both these cases.

Once I had produced an archive I had to check that it was a valid ZIP file, meeting the specifications of the ZIP file format (as described in Section 2). In order to do this, I used existing tools that recognise the ZIP file format. In this way, I could tell if my ZIP file had been produced incorrectly. If that was the case, trying to use the tools would produce an error, which allowed me to work out what was wrong in the way I was creating the archive, so I could fix it. I used the following tools:

- `less` to list the contents of the archive
- `vim` to open the archive and read the files stored inside
- `zipinfo -v` to display the details stored in the central directory
- `unzip` to unzip the archive
- `diff` to compare the unzipped file to the input file



```
Central directory entry #1:
---------------------------

  file.txt

  offset of local header from start of archive:   0
                                                  (0000000000000000h) bytes
  file system or operating system of origin:      Unix
  version of encoding software:                   6.3
  minimum file system compatibility required:     MS-DOS, OS/2 or NT FAT
  minimum software version required to extract:   1.0
  compression method:                             none (stored)
  file security status:                           not encrypted
  extended local header:                          no
  file last modified on (DOS date/time):          2020 Jul 28 09:56:22
  32-bit CRC value (hex):                         797b4b0e
  compressed size:                                57 bytes
  uncompressed size:                              57 bytes
  length of filename:                             8 characters
  length of extra field:                          0 bytes
  length of file comment:                         0 characters
  disk number on which file begins:               disk 1
  apparent file type:                             binary
  Unix file attributes (100664 octal):            -rw-rw-r--
  MS-DOS file attributes (00 hex):                none

  There is no file comment.
```

Figure 2: Partial output of the zipinfo command (verbose option).

Using `zipinfo` was especially helpful when populating the fields in the file headers and end of central directory record, to check that I had provided the correct values. Some example output of this command is shown in Figure 2.

5 Future work

Here I lay out the future steps to progress the work done in this project. The first step will be to make the ZipArchive API asynchronous, by using the asynchronous function calls from the XrdCl::File and XrdCl::Filesystem APIs, instead of the synchronous function calls currently in use.

Once that is complete, the ZipArchive class can be added into the XRootD project by combining it with the existing ZipArchiveReader class, found in *XrdClZipArchiveReader.hh*. In this way, there will be one class for dealing with ZIP archives, allowing both reading and writing. This will mean that the functionality to append files to ZIP archives will be available in the XRootD C++ API. After this, the append functionality can be added to the command line tool, `xrdcp`. Currently, it is possible to extract a file from a ZIP archive using the following command:

```
$ ./xrdcp --zip file.dat root://host//path/to/archive.zip root://host//path/to/destination
```

So, the idea would be to add an `--append` option, or similar, making it possible to append a file to a ZIP archive using the `xrdcp` tool in the following way:

```
$ ./xrdcp --append file.dat root://host//path/to/archive.zip
```

In order to account for the possibility of the append operation being interrupted, resulting in a corrupted ZIP archive with the EOCD overwritten, it will be necessary to implement a checkpointing mechanism in the XRootD server which will allow the recovery of the EOCD. A final extension to the ZIP support in the XRootD client would be to create a declarative API for the ZipArchive class. The current declarative API for the XRootD client can be found here.