



Evaluation of an Alternative Implementation to Native Operating System Extended File Attributes

by Joshua Edwards

ARL-CR-0686

December 2011

Prepared by
ICF International
401 E. Pratt Street, Suite 2214
Baltimore, MD 21202

Under Contract
W911QX-07-F-0023
Report No. 1057

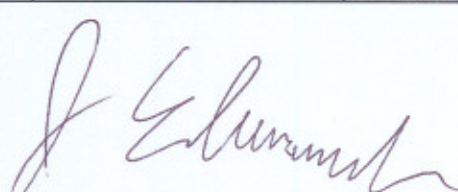
COR:
U.S. Army Research Laboratory
RDRL-CIN (Linda Duchow)
2800 Powder Mill Road
Adelphi, MD 20783-1197

ERRATA SHEET

**re: ARL-TR-5856 (now ARL-CR-0686), *Evaluation of an Alternative Implementation to Native Operating System Extended File Attributes*, December 2011,
by Joshua Edwards**

This is an errata sheet for ARL-TR-5856 (now ARL-CR-0686). The ARL number for this contractor report has been updated. Please attach this sheet to the cover page of the original document.

Page	Reads	Should Read
Cover	ARL-TR-5856	ARL-CR-0686 Prepared by ICF International, 401 E. Pratt Street, Suite 2214, Baltimore, MD 21202 Under Contract W911QX-07-F-0023, Report No. 1057 COR: U.S. Army Research Laboratory, RDRL-CIN (Linda Duchow), 2800 Powder Mill Road, Adelphi, MD 20783-1197
Title Page	ARL-TR-5856	ARL-CR-0686 Prepared by ICF International, 401 E. Pratt Street, Suite 2214, Baltimore, MD 21202 Under Contract W911QX-07-F-0023, Report No. 1057 COR: U.S. Army Research Laboratory, RDRL-CIN (Linda Duchow), 2800 Powder Mill Road, Adelphi, MD 20783-1197
ii	8. PERFORMING REPORT NUMBER: ARL-TR-5856	5a. CONTRACT NUMBER: W911QX-07-F-0023 7. PERFORMING ORGANIZATION NAME(S)/ADDRESS(ES): ICF International, 401 E. Pratt Street, Suite 2214, Baltimore, MD 21202 8. PERFORMING REPORT NUMBER: ARL-CR-0686 9. SPONSORING/MONITORING AGENCY NAME(S)/ADDRESS(ES): U.S. Army Research Laboratory, ATTN: RDRL-CIN-D, 2800 Powder Mill Road, Adelphi MD 20783-1197



Joshua Edwards
U.S. Army Research Laboratory,
Computational and Information Sciences Directorate, ATTN: RDRL- CIH-D,
Adelphi, MD 20783

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory

Adelphi, MD 20783-1197

ARL-CR-0686**December 2011**

Evaluation of an Alternative Implementation to Native Operating System Extended File Attributes

Joshua Edwards

Computational and Information Sciences Directorate, ARL

Prepared by
ICF International
401 E. Pratt Street, Suite 2214
Baltimore, MD 21202

Under Contract
W911QX-07-F-0023
Report No. 1057

COR:
U.S. Army Research Laboratory
RDRL-CIN (Linda Duchow)
2800 Powder Mill Road
Adelphi, MD 20783-1197

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188		
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) December 2011		2. REPORT TYPE Final		3. DATES COVERED (From - To)	
4. TITLE AND SUBTITLE Evaluation of an Alternative Implementation to Native Operating System Extended File Attributes		5a. CONTRACT NUMBER W911QX-07-F-0023			
		5b. GRANT NUMBER			
		5c. PROGRAM ELEMENT NUMBER			
6. AUTHOR(S) Joshua Edwards		5d. PROJECT NUMBER			
		5e. TASK NUMBER			
		5f. WORK UNIT NUMBER			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) ICF International 401 E. Pratt Street, Suite 2214 Baltimore, MD 21202		8. PERFORMING ORGANIZATION REPORT NUMBER ARL-CR-0686			
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: RDRL-CIN-D 2800 Powder Mill Road Adelphi MD 20783-1197		10. SPONSOR/MONITOR'S ACRONYM(S)			
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)			
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Extended attributes (xattrs) are a common means of storing file and directory metadata in a simple key-value pairing. Most, but not all, modern Linux file systems allow for this functionality. For those that do not, an alternative was developed using a locally stored database and a set of Python scripts to emulate the file system's method of storing xattrs. Comparing the two methods in three key areas—speed, size, and portability—will allow developers to determine which one may be the better option to fill their needs and fit their structure. The conclusion drawn favors using the database method in largely Python solutions and file systems that do not natively allow xattrs, while the file system method is preferred in most other situations or when hard drive space is in significantly short supply.					
15. SUBJECT TERMS Extended attributes, file systems, Linux					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 18	19a. NAME OF RESPONSIBLE PERSON Linda Duchow
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER (Include area code) (301) 394-1630

Contents

List of Tables	iv
Acknowledgments	v
1. Summary	1
2. Introduction	1
3. Procedures	4
4. Results	5
5. Discussion	6
6. Conclusions	7
List of Symbols, Abbreviations, and Acronyms	8
Distribution List	9

List of Tables

Table 1. Average completion time in seconds for different xattr methods and test values.	5
Table 2. Average completion time in seconds for a set of xattr functions with the number of simultaneous clients running.....	6

Acknowledgments

This research would not be possible without the code written and implemented by Travis Parker, an ICF International contractor for the U.S. Army Research Laboratory's (ARL) Computational and Information Sciences Directorate (CISD).

INTENTIONALLY LEFT BLANK.

1. Summary

Most modern Linux file systems use extended file attributes (xattrs) to store metadata for files in simple key-value pairs. This is generally done by enabling the libattr feature during kernel configuration and making calls to “getfattr” and “setfattr”. Recent work by Travis Parker, an ICF International contractor for the Computational and Information Sciences Directorate (CISD), enabled a new method for storing xattrs. This method uses a locally stored database (DB) and Python scripts to emulate the standard xattr functionality, while remaining largely file system (FS) agnostic.

This DB xattr functionality does not set out to wholly redesign the standard. It instead attempts to provide an additional option for certain scenarios. The benefits and tradeoffs become clear when comparing the two methodologies for speed, size, scalability, and portability.

The FS xattr methodology is the best all-around option. Since it is at the kernel level, it is fast and largely indifferent to the software calling it. The DB xattr methodology would be more appropriate in an environment where file systems may change. The DB method also shows slightly improved results in an all-Python environment where the xattr scripts can be imported and stored as an object, rather than making individual system calls.

2. Introduction

Extended file attributes are meant to augment owner, group, timestamp, and other traditional UNIX inode metadata. The functionality for xattrs rests in the libattr library and is compiled with the Linux kernel at build time. Linux uses xattrs internally to implement access control lists and labels, but they can also be used to store arbitrary user metadata. Depending on the attribute size, xattrs can either be stored within the inode space itself or as a pointer to additional data blocks^{*}. Since the xattrs are implemented at the kernel level, the setting and getting of xattrs are very rapid.

A kernel implementation is how xattrs are normally handled; however, an alternative exists. Travis Parker’s methodology detaches itself from the file system and uses a locally stored database and Python scripts to emulate the traditional xattr calls. This alternate method includes much of the same functionality, along with other enhancements that allow it to perform at a similar level in certain situations.

^{*} Hellwig, C. XFS: The Big Storage File System for Linux. *login* 34.5 (2009): 10–18.

The local database for this implementation consists of a single table, traditionally named “xattr”. This table holds three columns: “obj,” “attr,” and “val.” They are all standard text, varchar(255) columns that hold the full file path, the attribute key, and the attribute value, respectively. While any relational database is acceptable, our implementation uses PostgreSQL.

There are two major Python scripts involved:

- xa – This script is intended for use on the command line interface (CLI) as a single call and most resembles the FS xattr calls. Depending on the CLI arguments passed to it, it makes a connection to the database and reads, writes, or removes the xattr data for a file using calls to the imported dbXattr module. It then closes the database connection and ends.
- dbXattr.py – This is the primary script that xa and any other scripts use for all of the actual xattr-like functionality.

The dbXattr.py script is imported and instantiated as an object, dbXattr. The constructor allows for passing an already established database connection or the information to make its own connection. Once the database connection is established, the dbXattr script holds the connection open. Functionality also exists to add a file or directory, even recursively, to the cache from the start. A final flag can be set to allow it to automatically commit changes or hold a “dirty list” until the user orders it to commit the xattr changes.

After instantiation, individual functions can be called to set, get, and remove file xattrs using its respectively named functions. Each of the functions checks the internal cache to determine if the value is already stored. If it is not, it makes a call to the database to refresh its cache.

The following is an example of the get function:

```
def get (self,path,attr=None,scavenge=True):
    #see if we have it cached
    if path not in self.attrs:
        #if not try to fetch it
        try:
            self._cache(path,scavenge=scavenge)
        except Exception,e:
            print e
    [...]
```

The following is an example of the _cache function:

```
def _cache (self, path, recurse=False, scavenge=False,
autoCommit=False):
    cur = self.dbConn.cursor()
    if recurse:
        cur.execute(self.select+"obj LIKE '%s%'" % path )
    else:
        cur.execute(self.select+"obj = '%s'" % path )
    rows = cur.fetchall()
    cur.close()
```

```

if rows:
    for row in rows:
        if row[0] not in self.attrs:
            self.attrs[row[0]]={}
            if scavenge and not os.path.exists(row[0]):
                self.remove(row[0],autoCommit=autoCommit)
                continue
            self.attrs[row[0]][row[1]]=row[2]

```

Once the values are stored in the cache, the requested action is performed on the stored values in a dirty list variable. A get returns the cached value. A set creates or updates the cached value in a dirty list. A remove clears the value from the dirty list. If autocommit was activated or set to true as a function argument, it will alter the value in the database at the same time. If not, the user will need to call the commit function at a later time.

The commit function replaces the values stored in the database with the respective values in the dirty list. Objects that are not present in the dirty list are not altered in the database. The others are replaced with the new values using a SQL DELETE statement and INSERT statements.

The following is an example of the commit function:

```

def commit (self):
    cur=self.dbConn.cursor()
    if self.dirty:
        #remove all existing records, we should have all valid ones
        cached
        dsql=self.delete % ("'" + "','".join(self.dirty) + "'")
        try:
            cur.execute(dsql)
            for path in self.dirty:
                if path in self.attrs: #may not be there if we are
                    scavenging up stale records
                        for attr,value in self.attrs[path].items():
                            cur.execute(self.insert,(path,attr,value))
            except Exception, e:
                print e
        stat=self.dbConn.commit()
        if not stat:
            self.dirty=[]
        cur.close()
        return stat

```

The metrics compared for the two xattr methods are performance, size, scalability, and portability. Two experiments were conducted to evaluate performance and scalability, while size and portability are touched on in the discussion.

3. Procedures

The performance experiment was executed on a set of 5,000 flat text files. Four methods were used to set and get xattrs on these files, two for the FS method and two for the DB method:

- Individual calls to setfattr and getfattr (FS)
- The xattr Python library and its setxattr and getxattr functions (FS)
- Individual calls to xa (DB)
- An instantiation of dbXattr (DB)

A Python script was written to go through these methods and measure how long it takes to perform regular xattr functions. Each method would set and get its own pair of xattrs: a small, randomly selected 0 or 1 value and a larger value of 254 letter A's. The two values are primarily for the FS method to test speeds for a value that could potentially fit within a file's inode and a value that could not. The inode size for the test system was 256 bytes.

The instance of dbXattr was created at the start of the script, and that object was used for all of the relevant tests. A locally installed PostgreSQL database dedicated to the experiment was created for the two DB xattr attempts.

Time was measured using Python's standard time module. The CLI methods, getfattr/setfattr and xa, were called using Python's subprocess module and its call() function. The xattr library for Python is used to give a more accurate view of time taken with the FS method. The FS method is more direct than using a subprocess class as it does not need to set up a new shell instance with each call. The subprocess calls to getfattr and setfattr are retained to better balance time taken when comparing the FS method to xa calls.

The scalability experiment was set up similarly, except on a system with a 128-byte inode size and using an external, networked database. Its primary goal was to determine how well the DB method performed with multiple simultaneous clients running and on an already overloaded file system. It used three methods for manipulating xattrs on separate sets of 5,000 flat text files:

- Calls to the xattr Python library (FS)
- The same calls to the xattr Python library while the file system is overloaded with constant external calls (FSO)
- An instantiation of dbXattr with caching unused (DB)

The FS and DB method each have their own "client" script, while the FSO method mirrors the FS script. Both clients begin by creating a unique set of 5,000 flat text files. Afterward, they enter a

constant loop that performs a set of xattr actions: sets a small and a large pair of xattrs akin to the performance experiment, reads each pair for each file 10 times in a random order, and finally removes all of the xattrs. Once it completes a set, the time elapsed while running the set is presented.

The FSo method uses the same client as the FS method; however, while running, several input/output (I/O) overloader scripts run in the background. These scripts simply copy and remove large text files and are meant to flood the file system with I/O requests, simulating a busy server. This is to determine if there is any significant performance degradation between the methods as use is scaled up.

The performance was measured for the three methods with different numbers of concurrent clients running. We set a baseline with one client and then collected times for 20, 30, 50, 75, and 90 simultaneous clients.

4. Results

From the performance experiment, we found that the DB method generally performed worse. When used as standalone CLI calls, the DB method performed roughly 20 times slower than the standard FS method using subprocess calls, and around 1500 times slower than the direct xattr library calls. When instantiated within the Python script itself, however, the RAM caching and always-on database connection gave an occasional minute speed boost over the FS method, when retrieving values (table 1).

Table 1. Average completion time in seconds for different xattr methods and test values.

Test	Time
DB small xattr set time	194.878011
DB (cache) small xattr set time	4.007993
FS small xattr set time	9.737148
FS (direct) small xattr set time	0.129128
DB small xattr get time	186.471242
DB (cache) small xattr get time	0.009079
FS small xattr get time	9.880415
FS (direct) small xattr get time	0.102592
DB large xattr set time	210.681424
DB (cache) large xattr set time	5.04396
FS large xattr set time	11.969459
FS (direct) large xattr set time	0.132928
DB large xattr get time	202.243332
DB (cache) large xattr get time	0.008986
FS large xattr get time	12.377902
FS (direct) large xattr get time	0.008148

Regarding disk space, the PostgreSQL data directory took up 79,736 kilobytes prior to testing. After the testing was completed, the data directory grew to 81,896 kilobytes. Thus, the space needed to hold the four sets of xattr data for the DB method, 2,000 rows, was 2,160 kilobytes.

The scalability experiment had similar results. All of the methods showed longer run times as the number of clients increased; but, the load never became so drastic that the DB method surpassed the FS methods. Table 2 shows the average completion time in seconds for a set of xattr functions with the number of simultaneous clients running. The numbers between the performance and scalability experiments likely differ due to the different hardware configurations.

Table 2. Average completion time in seconds for a set of xattr functions with the number of simultaneous clients running.

Clients	FS	DB	FSo
1	3.06	21.23	3.36
20	9.93	45.53	12.04
30	14.68	60.09	19.85
50	24.96	116.99	37.79
75	38.67	176.6	37.21
90	44.87	243.43	42.12

5. Discussion

Not all Linux file systems support xattr functionality, e.g., Network File System (NFS). Python and a separate database were used specifically to allow more portability. For this purpose, the databases are indifferent to the file system used and only require the ability to communicate with each other. The method varies depending on the database used. In the experiment with PostgreSQL, the psycopg2 Python module was used as the interface with the database.

The caching, while giving similar speeds to standard xattrs, has its own pitfalls. A stored cache is not regularly updated unless forced. Other processes updating the same file's xattrs would introduce asymmetries between the cache and database. This can produce unintended behavior for long-running daemons if they rely on monitoring xattrs for changes. This issue can be mitigated by forcing the cache to be refreshed after a set time. The only way to eliminate this issue is to deactivate caching. The speed is lost, but the integrity is maintained.

The possibility of data loss also exists when using a dirty list without autocommit to store xattrs and the running process fails. These values remain in volatile memory until stored with a function call.

6. Conclusions

The optimal implementation for xattrs depends on system limitations and the desired effect intended by the administrator. In terms of space usage, the FS method would be the most viable option. By setting values within the white space of inodes, there very well may be no additional space taken at all. However, in this age of ever-expanding hard drives, this should be considered an issue only for the most space-constrained circumstances.

The performance test results indicate where the two xattr methods would be best suited. The FS xattr method is a good, all-around option that is fast enough for most needs. A standalone CLI call to `xa` is noticeably slower. In a largely Python-based environment, however, the increase in speed from caching and the opportunity to instantiate an object present in the DB method may be more attractive than using subprocess calls or a similar wrapper.

The performance degradation was fairly consistent as the load on the hardware increased in scale for both methods. The FS method continued to outperform the DB method, even when the file system was overloaded.

For Linux file systems that do not support xattr functionality, e.g., NFS, the DB method of storing xattrs would be an option. Most of the standard CLI functionality is emulated within the `xa` script. It is also portable; data written can be easily transferred from one file system to another.

List of Symbols, Abbreviations, and Acronyms

ARL	U.S. Army Research Laboratory
CISD	Computational and Information Sciences Directorate
CLI	command line interface
DB	database
FS	file system
FSO	file system overloaded
I/O	input/output
NFS	Network File System
xattr	extended attribute

1 DEFENSE TECHNICAL
(PDF INFORMATION CTR
ONLY) DTIC OCA
8725 JOHN J KINGMAN RD
STE 0944
FORT BELVOIR VA 22060-6218

1 DIRECTOR
US ARMY RESEARCH LAB
IMNE ALC HRR
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 DIRECTOR
US ARMY RESEARCH LAB
RDRL CIO LL
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 DIRECTOR
US ARMY RESEARCH LAB
RDRL CIO MT
2800 POWDER MILL RD
ADELPHI MD 20783-1197

6 DIR USARL
RDRL CIN D
J EDWARDS (3 CPS)
M SHEVENELL
P GUARINO
C ELLIS
2800 POWDER MILL RD
ADELPHI MD 20783-1197

INTENTIONALLY LEFT BLANK.