I. J. Computer Network and Information Security, **2016, 11, 32-43**
Published Online November 2016 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijcnis.2016.11.04

FileSyncer: Design, Implementation, and Performance Evaluation

Oluwafemi Osho
Department of Cyber Security Science Federal University of Technology, Minna
E-mail: femi.osho@futminna.edu.ng

Anthony Ugbede Faruna
E-mail: thony4u@gmail.com

***Abstract*—**With the pervasiveness of information technology, one of the growing trends today is a phenomenon which can be termed one-user-to-many-computing-devices. In many cases, the need to manage information across multiple electronic devices and storage media arises. The challenge therefore is finding a file synchronization system that can effectively replicate files across these different devices. This paper presents the design, implementation, and evaluation of FileSyncer, a rapid and efficient file synchronization tool that, in addition to the traditional synchronization capabilities, supports manual update selection and mechanism to revert a synchronization process back to the last previous state. The system employs last modified time, file size and CRC checksum for update detection and to ensure integrity of synchronized files. The synchronization times of the system for files of different sizes were compared with those of four existing file synchronization systems. Results showed increased efficiency in terms of time taken by FileSyncer to complete a synchronization operation with increase in file size compared to the other systems. In the future, we plan to release FileSyncer to the open source community for further development.

***Index Terms*—**File synchronization, backup, recovery, update detection, reconciler, security.

I. Introduction

The rapid growth of information technology and mobile computing has brought to light the need for maintaining copies of data in different locations [1]. The growing trend of many computing devices to one user has come with huge expectations of replicating files across these different devices. Issues in data management like loss or damaged data have also highlighted the need for a backup strategy to recover data when such incident arises. Thus, the rise in the need to manage information across multiple electronic devices and storage media used by information users poses the challenge to develop an easy to use and efficient file synchronization system.

A file synchronization system helps to maintain consistency of the state of two computer files in different locations by constants updates, using a defined algorithm [2]. It is essential that any file synchronization tool would be easy to use, and flexible enough to restore or revert back a synchronization operation in the case where the synchronization does not yield the required or desired result.

File synchronization is applicable in virtually all facets of information technology, including cloud [3], [4], distributed network [5], [6], [7], peer-to-peer network [8], [9], and video [10].

Primarily, a file synchronization system is made up of two components, namely update detector and reconciler [11]. Typically, the update detector analyzes and automatically detects changes made to the file replicas since the last synchronization operation. The reconciler then combines the update to yield the new synchronized and updated state of each of the file replica. These steps are typically repeated each time the synchronization is initiated. However, there are instances where a user may probably want to select files to be synchronized. This happen more often in a situation where the updates to be propagated are minimal, in a large volume of files. Using the traditional way of synchronization will not be effective in this case, considering the needed update detection time and significant CPU cost that would be incurred. As an example, a user has 1000 main folders, with each containing sub-folders and files. Since the last synchronization, he has only made changes to the sub-folders and files in just one of the main folders. A good solution is for the synchronization system to provide mechanism for such user to manually select that one folder to be synchronized, without subjecting the entire folders to the automatic update detection, and then move directly to reconcile the replicas. It therefore becomes necessary to have a system that, in addition to providing mechanism for automatic update detection, allows the user to manually select files to be updated, thus, skipping the automatic update detection. In this case, significant time can be gained.

This study focuses on the development and evaluation of FileSyncer, a rapid and efficient file synchronization tool that supports fast synchronization, manual update selection and mechanism to revert a synchronization process back to the last previous state.

The rest of the paper is organized as follows: sections two, three, and four provide extensive literature review of file synchronization system. Specifically, they discuss the

taxonomy, components, and limitations of file synchronization system respectively. In section five, the methodology is presented. The system design, implementation and testing, and evaluation of the performance of the system are covered in sections six, seven, and eight in the order given. Lastly, in section nine, the study is concluded.

II. Taxonomy of File Synchronization System

File synchronizers can be classified on the basis of direction of the synchronization operation and on the way conflicts are treated. Most file synchronizers propagate updates just in one direction, while others propagate in two directions. It is also noteworthy that every file synchronizer by default propagates updates that are not conflicting. The basis for propagating conflicting updates depends on the specifications and methods used by the file synchronizers. Fig. 1 provides a diagrammatic depiction of the taxonomy of file synchronization.

A. File Synchronization Based on Direction

A file synchronization based on direction focuses on the direction of the synchronization operation. Based on the direction, synchronization can either be propagation of updates from source directory to a target directory or from a target directory to a source directory, or it could be a simultaneous propagation between the source and target directory. These are termed one-way synchronization and two-way synchronization respectively [11], [12], [13].

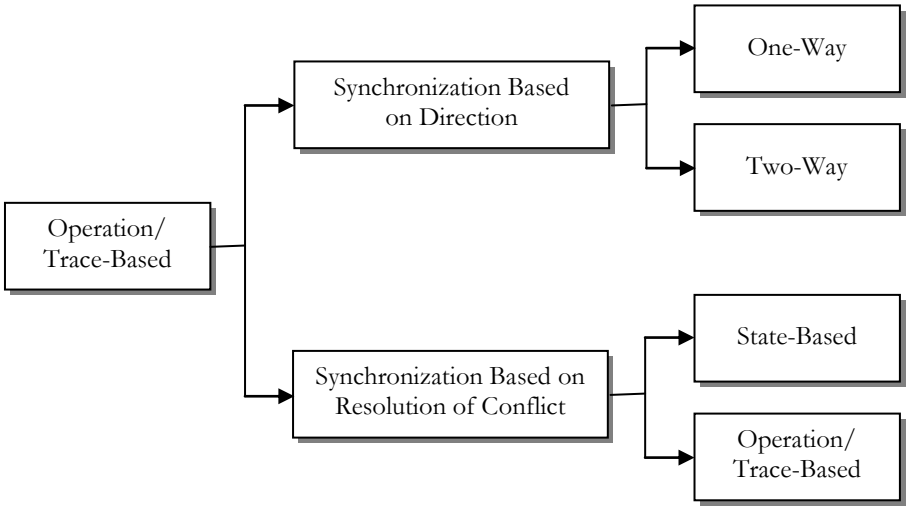

Fig.1. Taxonomy of File Synchronization System

One-Way File Synchronization

This type of synchronization propagates changes made in one direction, and the contents are expected to change in only one replica. The synchronization operation reconciles the changes made in only one of the replicas. The replicas cannot be really considered to be in the same state here since the updates are not being propagated in both directions. One of the replicas is referred to as the source while the other as the target. The updates are propagated from source to target only. Take for example, a case where a new file was added to a replica A (Source). The update will be propagated and reconciled to the replica B (Target). If we also have a new file in replica B, it will not be propagated to replica A.

Two-Way File Synchronization

This type of synchronization operation propagates updates of files in both replica and directions, to ensure consistency, and reconcile changes made in both ways. The contents of the replicas are expected to change in the different locations. Both locations would have the same state as a result of the synchronization operation. For example, if there was a file that is new in a replica A, A will be propagated to replica B. And if replica B also contained a new file, it will be equally propagated to replica A.

B. File Synchronization Based on Resolution of Conflict

File synchronization system performs synchronizations based on resolution of conflicts mechanism. Under this category, file synchronization systems are further sub-categorized as either state-based file synchronizers or operation/trace-based file synchronizers [1], [2], [11], [14].

State-Based File Synchronizers

These synchronizers utilize the present state or contents of the filesystem to detect updates. This includes observing modification times, inode number, dirty bits, contents of files, and comparing them against the copies that was saved. State-based synchronizers tend to be very portable; does not require administrative privileges, which makes them suitable for use as user-level programs; and are readily used in situations where it is very impractical to use full-blown distributed filesystems and databases. A major issue associated with state-based synchronizer is how to determine the areas changes and

modification were made by aligning the present replicas and the replica that was saved at the last synchronization operation.

Operation/Trace-Based File Synchronizers

Operation/trace based synchronizers detect possible updates by examining sequence of all operations carried out on the replica and all the modifications and changes made to the file. The operating system provides the trace to the file synchronizer when it is being executed or it can also monitor the updates and changes being made in real time. Although operation-based synchronizers possess more detailed information to make decision, which implies a better decision in regards to propagating changes and resolving conflicts, they, however, need the support for building the synchronization process in the system at a very low level.

III. Components of File Synchronization System

Every file synchronization system, as depicted in Fig. 2, is basically made up of two components, which are the update detection mechanism and the reconciliation algorithm [1], [2], [10], [11], [14], [15], [16], [17], [18].

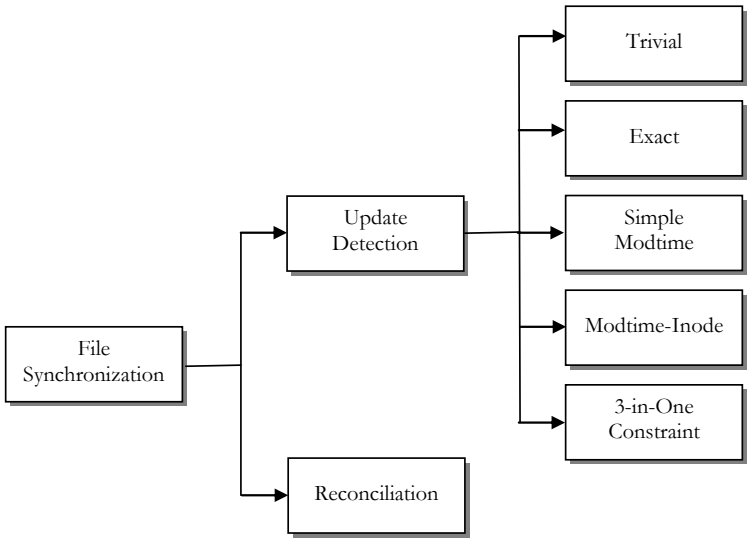

Fig.2. Components of a File Synchronization System

A. Update Detection

The update detector of each of the file replica S calculate a $dirty_S$ that will summarize all the changes that have been applied to S. While there is room to make error for safety sake to indicate possibility of changes where none happened, it is necessary that real changes made must be detected [1].

Both filesystems in each of the replicas are the same initially with contents O. When a user updates one of the replicas, or both, it results in new states of A and B. During the synchronization operation, using either the size or modification time of the two replicas, the update detector detects changes on both replicas and calculate the predicate $dirty_A$ and $dirty_B$. These predicates with the current state are used by the reconciler to compute the new states. The following specifications are made for the update detector [1], [2], [11]:

Definition 1: We define a predicate *dirty* as an up-closed predicate on the file paths, i.e., predicate ϕ such that, if we have p ≤ q and ϕ(q), then $\emptyset(p)$. The implication of this statement is that if $\emptyset$ is predicate dirty, then $p \leq q \wedge \neg\emptyset(p)$ implies $\neg\emptyset(p)$.

Definition 2: Assuming that O and S are filesystems and $dirty_S$ is a dirtiness predicate. We can then say $dirty_S$ is to calculate the updates from O to S if $\neg dirty_S$ (P) means O/p, given all paths p. One important feature of this definition is that if A, B, and O are filesystems and $dirty_A$ and $dirty_B$ computes the update from O to A to B, then $\neg dirty_A(P)$ and $\neg dirty_B(p)$ together imply A/p = B/p.

The update can be implemented with different strategies. These include [1], [2], [11]:

Trivial Update Detector

This is one of the simplest implementations which present a predicate that is always true; it just denotes all file as dirty, with the reconciler taking every file, except the ones that are the same in the two filesystems, as conflicts. This can be a fairly acceptable update detector strategy for situations where the filesystems are small, but it becomes an issue, in large filesystems, as the reconciler has to compare all the files in both filesystems.

Exact Update Detector

This implementation strategy differs slightly from the trivial update detector, as it exactly computes the dirtiness predicate for the replicas by storing a duplicate

of the entire filesystem during the last synchronization operation, and comparing that copy with the current filesystem. To detect update in an exact style is quite expensive in terms of the space it will take to keep the former copy of the filesystem, and more notably, the time it will take to compute the changes made to the present contents with the copies of the filesystems that were saved. Although this implementation can perform well when there is adequate time to carry out the synchronization operation.

Simple Modtime Update Detector

This implementation for the update detector is much cheaper but accuracy is on the low side. The update detector uses the last modification time to compute changes that were made to the filesystems. In order to detect the changes, the last modification time of each file replica is assessed against its value and checked whether it is old or new, and marked as dirty if it is new. Although this strategy is quite simple but it is not just enough to detect changes by only looking at the file modification time and its directories, since its location can be changed, leaving its modification time alone but altering its parent directory modification time. To handle and prevent this problem carefully, we call a file dirty given that at least one of its parent directories has a more recent modification time than the last synchronization operation.

Modtime-Inode Update detector

A more efficient strategy of recognising updates under UNIX operating system is the use of the modification time and inode numbers. We not only make use of the last time of synchronization but also the inode number of every single file in each copy. A path will be marked dirty by the update detector given that its inode number is different from the one stored or the modification time is different from the time of the last synchronization.

3-in-One Constraint Update Detector

This type of update detection mechanism combines three different constraints/merits to detect and propagate updates during synchronization. This method does not only employ the last modification time but also makes use of the size of the file and CRC (Cyclic Redundancy Check) checksum of the files that needs to be updated.

CRC is a type of algorithm known as a hash. A hash algorithm accepts variable-length input and produces a fixed-length output which uniquely represents the input data. The hash is usually much shorter in length than the data it represents. A sample CRC value for a file could be DAF42G8R. In theory, no other file should produce the same hash value. The file synchronizer calculates the CRC value of one file and compares it to the CRC of the corresponding file in the other folder. If the CRCs differ, the files differ.

B. Reconciliation

The reconciler works by using the predicates to determine which replica is most recent and contains changes. The process of combining this updates from the various replicas to yield a new synchronized state is called reconciliation [2].

The following are the definitions for a reconciler as follows [1], [2], [11]:

Definition 1: We have two filesystems A and B. A given path p is said to be useful in (A, B) iff either $p = \varepsilon$ or $p = q.x$ for some q and x, with A (q) = B (q) = DIR.

Definition 2: The new pair of the filesystems (C and D) is given as the result of the synchronization of the primary filesystems A and B with regards to the predicate dirty for both A and B. The following conditions are satisfied for each path relevant in A, B.

- $A(p) = B(p) \Longrightarrow C(p) = A(p) \wedge D(p) = B(p)$
- $\neg dirty_A(p) \Longrightarrow C/p = D/p = B/p$
- $\neg dirty_B(p) \Longrightarrow C/p = D/p = A/p$
- $dirty_A(p) \wedge dirty_B(p) \wedge A(p) \neq B(p) \Longrightarrow C/p = A/p \wedge D/p = B/p$

IV. Limitations of File Synchronization System

Several challenges for file synchronization systems have been identified. These includes: the problem of set reconciliation, the design and semantics of a file synchronization system, CPU and bandwidth optimization, and communication complexity [1], [16], [17], [19], [20], [21].

A. The Problem of Set Reconciliation

One challenge faced by most implementations of remote file synchronization system is having to avoid sending the whole file during the synchronization operation. For files containing large group of little register, e.g., schedule on a mobile device, the issue is how to efficiently detect files that have been modified without transferring specific fingerprint or time stamp for each files. This issue is termed as the problem of set reconciliation. Most present file synchronizers sends all the files if any file has changed which is fair enough for record-based data that are small, but not in the case of files that are large. To say the least, there exist the non-trivial issue of properly stating the semantics for the file synchronization system.

B. The Design and Semantics

The process of designing a file synchronization system is tedious and a demanding goal. The basis for this is the fact that file synchronization system must deal with every details regarding the semantics and low-level twists of real-world filesystems. A file synchronizer software that naturally handles operation in a distributed fashion is expected to be proactive in the face of possible host and network failure. This is so because misbehaviour of a file synchronizer can damage and corrupt random user files and data. Thus, correctness of the file synchronizer

design becomes a critical issue.

C. CPU and Bandwidth Optimization

Several issues often occur in trying to optimize the amount of bandwidth consumption. There are various types of overhead that may arise in some situations during the process of remote file synchronization. The first is the cost of CPU incurred due to calculation of hash and structure of data lookups and insertions. The second issue is the expenses incurred in scanning the file system and files retrieval. The latter cost is mostly substantial when ensuring consistency of large directory trees with few updates and modifications, and is the same for most techniques used to perform remote file synchronization.

D. Communication Complexity

Another file synchronization limitation is the present communication bounds for feasible protocols, which are still a logarithmic factor from the lower bounds for most interesting distance metrics, even for multi-round protocols

V. Methodology

The major objective of this work is the development of FileSyncer, a file synchronization system that provides rapid synchronization, offers user the ability to manually select specific files and path to synchronize, and also ensures the possibility of restoring to a pre-synchronized state.

Information system design methodology approach was utilized in the development of the system. Essentially, it is made up of four distinct phases: planning, analysis, design and implementation [22].

VI. System Design

We present, in this section, the design of FileSyncer. The functional and non-functional requirements are defined. In addition to this, the framework of the system is discussed. The structures of the different components are equally discussed.

A. Requirements Definition

The requirement of a file synchronization ranges from detecting conflicting and non-conflicting updates, and reconciling updates in a timely manner. The safety of file synchronization operation in terms of treatment of conflicts is an important issue to be considered and given top attention during development life cycle.

Non-Functional Requirements

The following are the basic requirements of our file synchronization system:

- Ease of Use: The file synchronizer software should have a friendly and interactive interface that is not complex for a user to operate.
- Openness: The file synchronizer should be open to contributions and improvements by third party individuals. This will help to improve, to a large extent, on errors that may be present in the system.
- Safety: The file synchronizer should ensure appropriate and correct synchronization. The synchronizer should not take arbitrary decisions without the knowledge of the user, as this may result in an undesired synchronized state of files.
- Robustness: The synchronization software should have the ability to handle possible errors during synchronization operation in a timely and orderly fashion.

Functional Requirements

The functional requirements of a file synchronization system are those core and basic requirements needed to perform a synchronization operation. They mainly include detection of updates, reconciliation along with other features. These requirements are itemised below:

- The file synchronizer should be able to detect updates where changes were made in any of the file replicas.
- The synchronizer should be able to perform synchronization from source to target or vice versa, as the case may be.
- The file synchronizer should perform synchronization in both directions, that is, it should be able to detect and reconcile updates in the different replicas.
- The file synchronizer should provide mechanism to restore from new state to old state by offering a backup option before performing any synchronization operation.
- The file synchronizer should offer the ability to preview changes before it is made.
- The file synchronizer should provide an audit log, containing synchronization details, after every synchronization operation.

B. Framework for FileSyncer

FileSyncer is a state-based file synchronizer that is consisted in a 3-in-one constraint update detector, and reconciler. Fig. 3 presents the process of file synchronization.

The system is essentially an application for synchronization of files between different media, including computers and storage media. Since it does not involve remote connection, the issue of minimizing bandwidth is consequently eliminated, as the file synchronizer only detects and reconciles local changes.

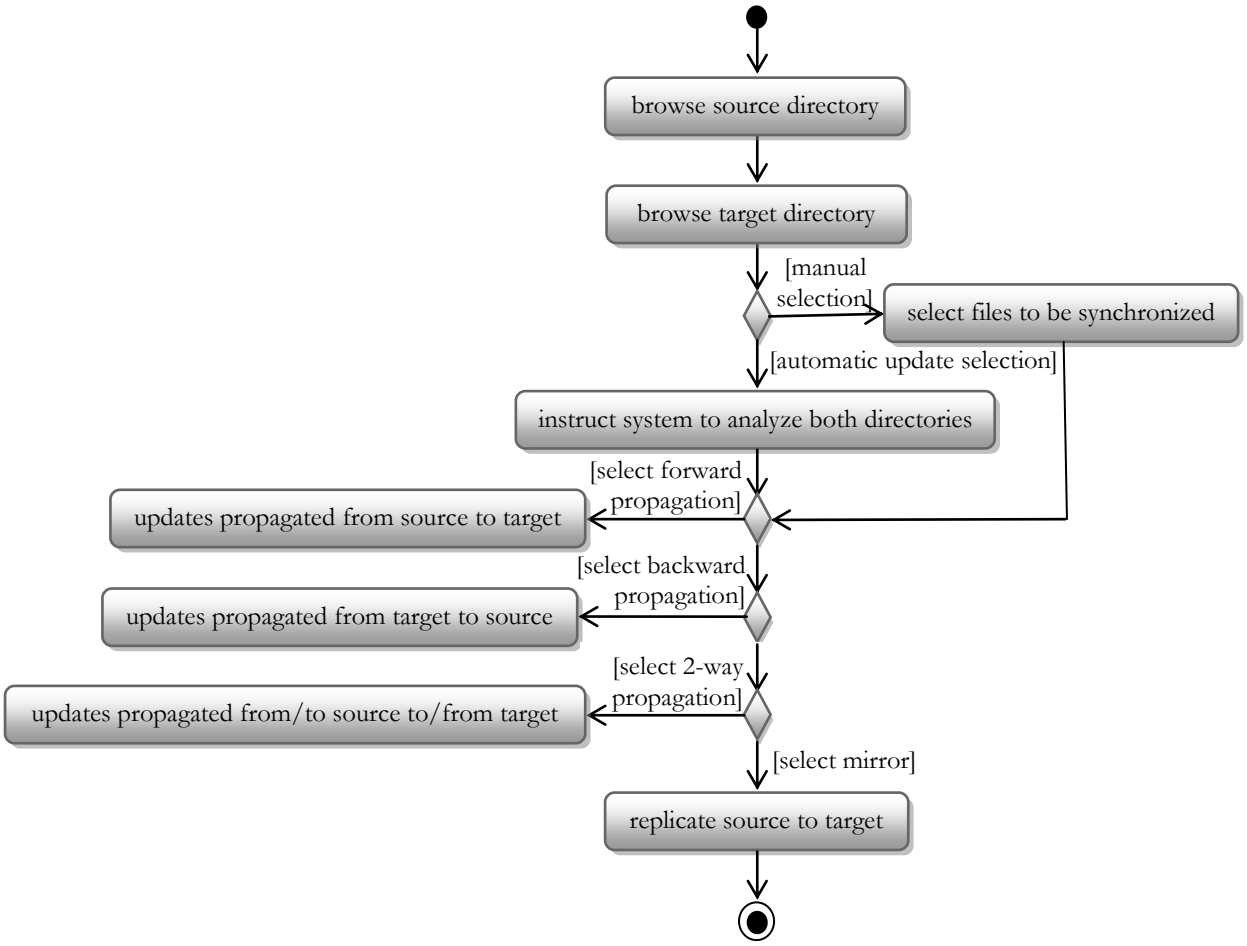

Fig.3. Activity Diagram of Syncrhonization using FileSyncer

Two special features of FileSyncer are functionalities to manually choose files to be synchronized and revert to previous versions of files synchronized.

C. Update Detector

The update detector for the file synchronization is based on three constraints which are Cyclical Redundancy Checksum (CRC), file size and file modification time.

The update detector will therefore detect update by comparing the two files to be synchronized using any of the three metrics to know where there was an update. An update is said to be detected if the last modified time for both files are different, the file size for both files are different or there is difference in the checksum calculated for both files. Depending on the mode of synchronization, these updates are propagated either from source to target, target to source, or simultaneously between source and target.

When performing forward or backward file synchronization, the file synchronizer works by comparing the file modification time of the source and target directory to propagate update. For the forward synchronization, the source directory is marked as the directory with the latest modified time. On the other hand, when performing backward synchronization, the target directory is marked as the directory with the latest modification time.

For a two-way synchronization, the file synchronizer makes use of file size to propagate updates. The detector compares both directories, and detects that there were updates made to both, thus resulting in them having different file sizes. The update detector will simultaneously propagate the updates that were present in one file to the other, and vice versa.

The detector also makes use of CRC checksum, as a metric, to ensure the integrity of the synchronization operation. The checksum values for both source and target directories using CRC algorithm are computed. This guarantees that there are no errors during synchronization.

D. Reconciler

Reconciliation is the process of combining the updates and changes made to the various file replicas to yield new consistent replicas that are identical and the same. Based on the updates detected by the update detector, FileSyncer implements a reconciliation algorithm which propagates these updates to get new states. The specifications of the reconciler supported by FileSyncer are forward, backward, and two-way synchronization, and mirroring. These are modelled using Fig. 4 to 6.

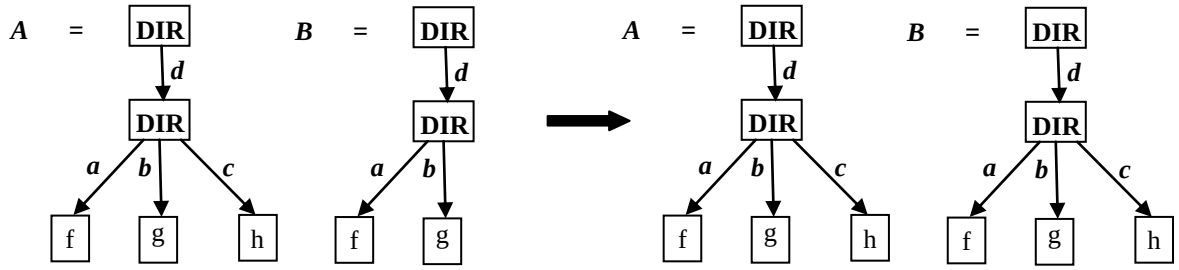

Fig.4. Forward/Backward Propagation of Updates

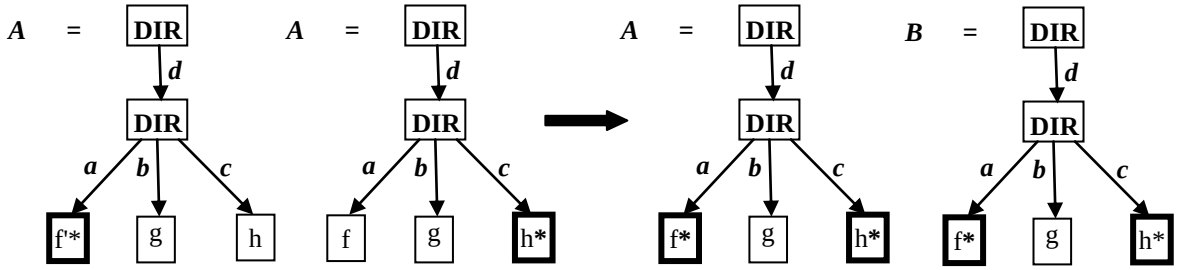

Fig.5. Two-way Propagation of Updates

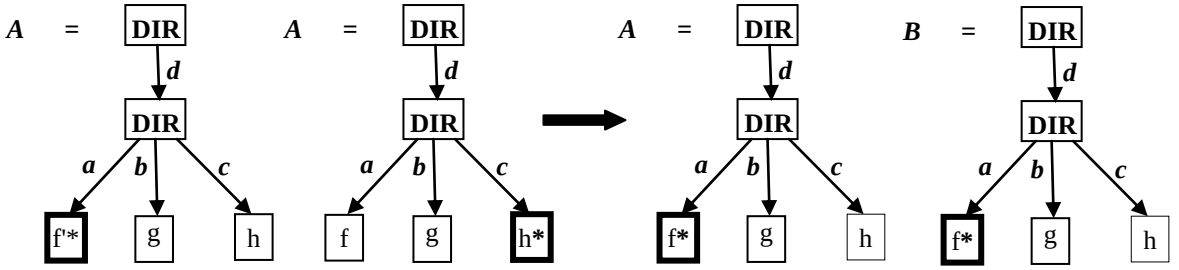

Fig.6. Mirror Propagation of Updates

E. Restore Operation

To support restoring of files to their pre-synchronized states, the system provides an option to back up files before performing the synchronization. The backed up files are compressed and placed in a zip folder. This helps to reduce the size, to conserve memory. The zip folder is stored in a directory on the user's system. The file synchronizer maintains a database that stores all the files that were backed up to enable retrieval.

VII. System Implementation and Testing

The system was implemented using Java, on NetBeans 8.0.2. The choice of Java was based on its distinct features. Apart from being object-oriented, it is platform independent, simple, reliable and secure, and provides well-designed intuitive set of APIs which helps programmers write better code with fewer bugs. Fig. 7 presents the main interface of the system.

At the top of the software interface is the Log button, used to display the details or summary of the last synchronization operation. There are two Browse buttons which are used to select the Source and Target directories respectively. The Analyze button is used to display list of files that needs to be updated and propagated. It checks both source and target directories, and identifies the files that were modified in either directory.

The forward arrow Sync button is used to perform synchronization from source to target regardless of changes that may be present in the target directory. The backward arrow Sync button performs synchronization from target to source, also regardless of changes that may be present in the source directory. It only propagates changes in the target directory. The backward and forward Sync button represents a two-way synchronization operation, for propagating updates and changes made to both source and target. This synchronization method is mostly used when the user is sure to have made changes to both source and target directories at different times.

The next is the Mirror button, which helps to replicate the contents of the source directory exactly to the target directory. This option is used, for instance, when a user rearranges a directory and may want to synchronize with the target directory, even though there are no updates to be transferred.

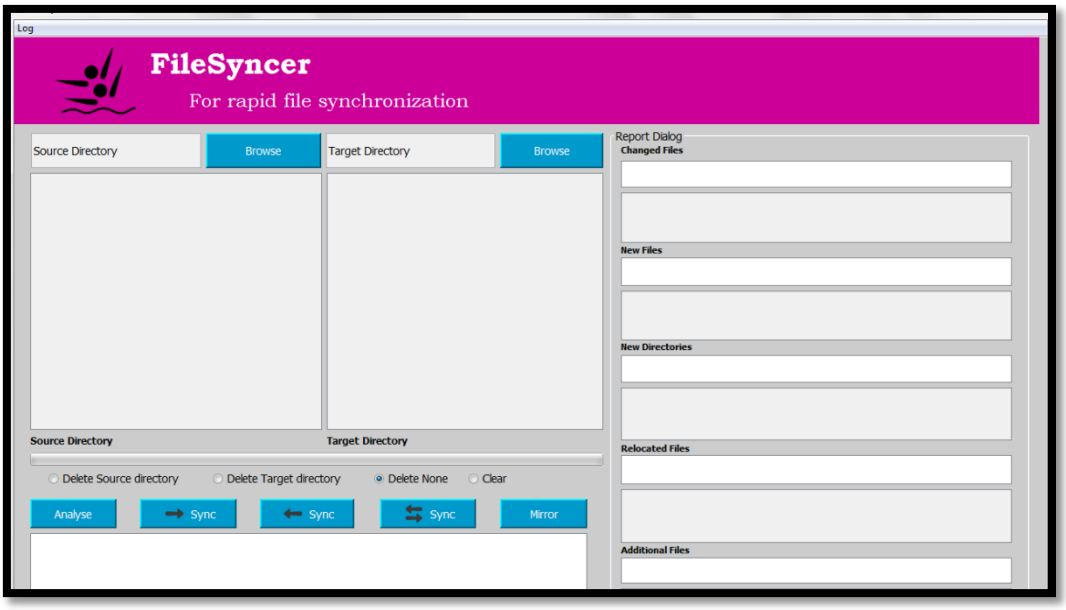

Fig.7. Main Interface

The Delete Source directory button is used to delete a source directory that may no longer be needed, while the Delete Target directory button is used to delete a target directory from its location. The Delete None operation preserves both source and target directories. The Clear button clears the details of a synchronization operation..

The right hand pane of the software contains the Report Dialog panes. The panes display information regarding the synchronization operation, including the changed files, new files, new directories, relocated files and additional files. The Changed Files pane displays information regarding files that were changed and propagated. The New File pane shows information on files that were added to either source or target directory, and were propagated. The New Directories pane provides details on new directories that were created and propagated, Relocated Files pane information on all files that were synchronized, that were actually moved from one folder/directory to another, and Additional Files pane information about files that may have been propagated even though they probably existed in another location in the source or target directory.

The capability of the system to manually indicate files to be synchronized is demonstrated in Fig. 8 and 9. From the left and right directories, four and two files respectively are indicated to be synchronized.

Using two-way synchronization, the right directory is updated with the four selected files from the left directory, while the left directory with the two files selected from the right directory.

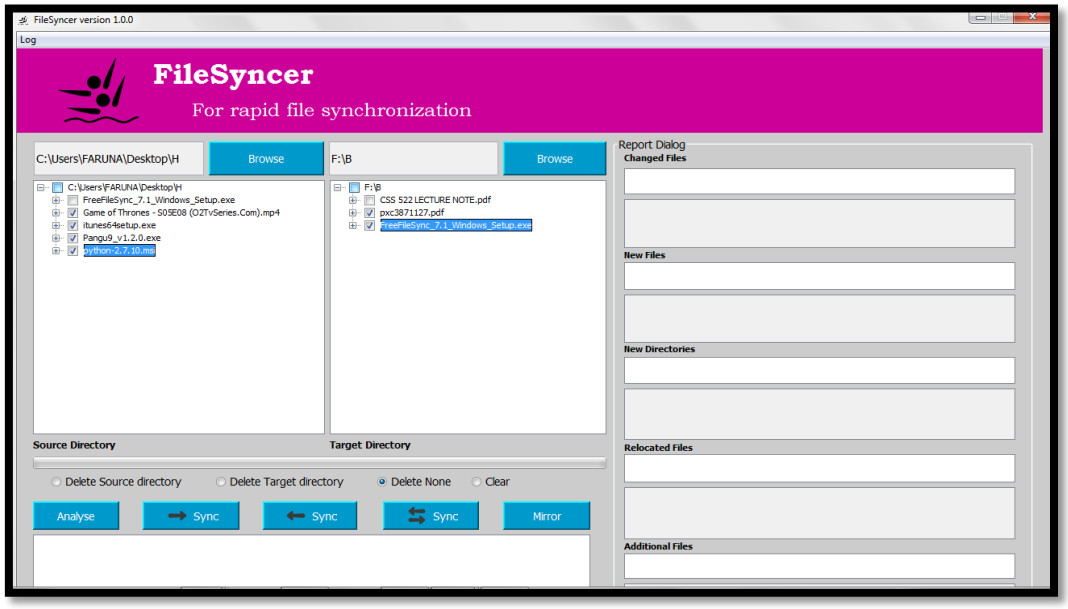

Fig.8. Manual Selection of Files to be updated from Both Directories

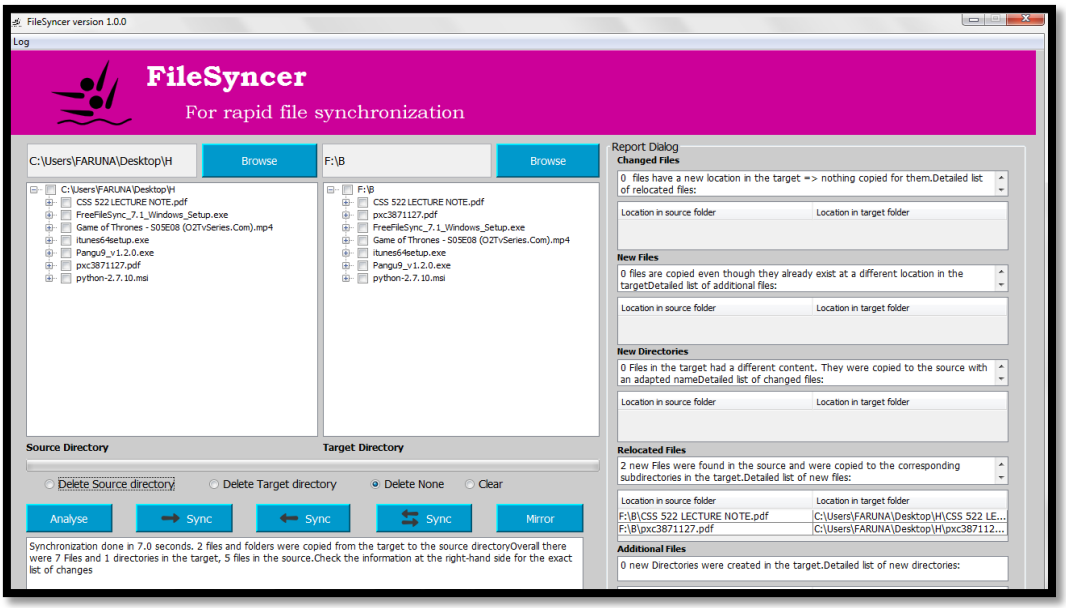

Fig.9. Two-way Synchronization of Files Manually Selected

VIII. Performance Evaluation

To evaluate the performance of FileSyncer, two tests were performed: system integrity and synchronization speed.

The integrity of the system is measured in terms of its capacity to synchronize files without modifications to the integrity of the file. To determine this, we calculate a type of checksum known as MD5, using FastSum, a tool developed on the basis of the generally accepted MD5 checksum algorithm which is used globally for assessing and testing the integrity of files [23]. The integrity of each file in the source and target directories, after a synchronization operation, is determined. Results of the integrity tests are presented in Fig. 10 and 11.

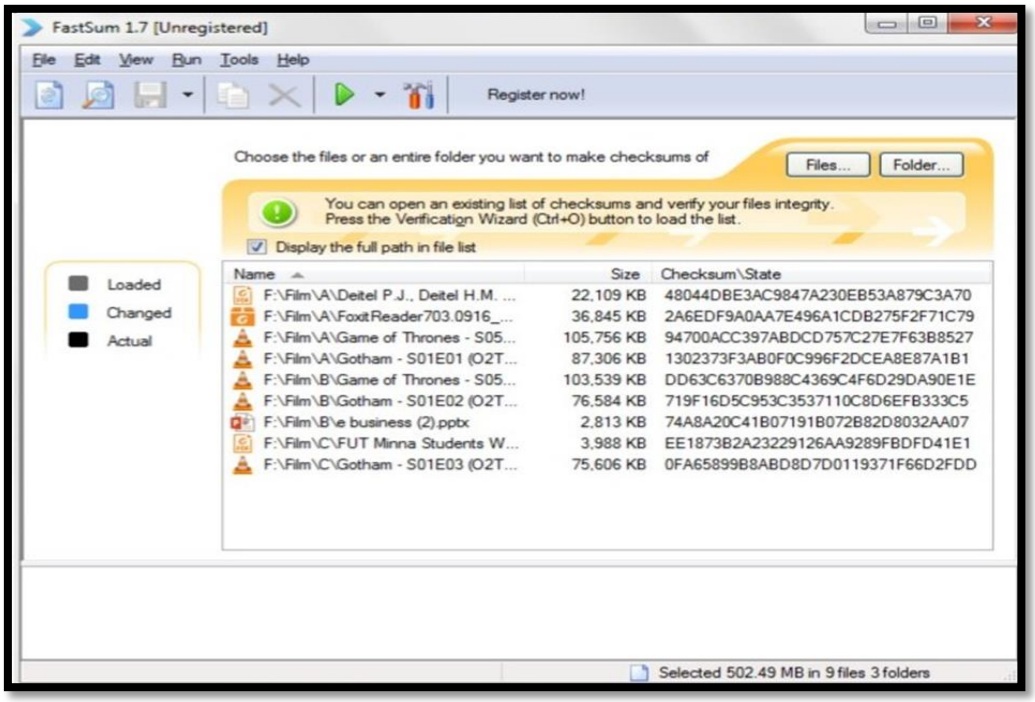

Fig.10. Integrity Test of Source Directory

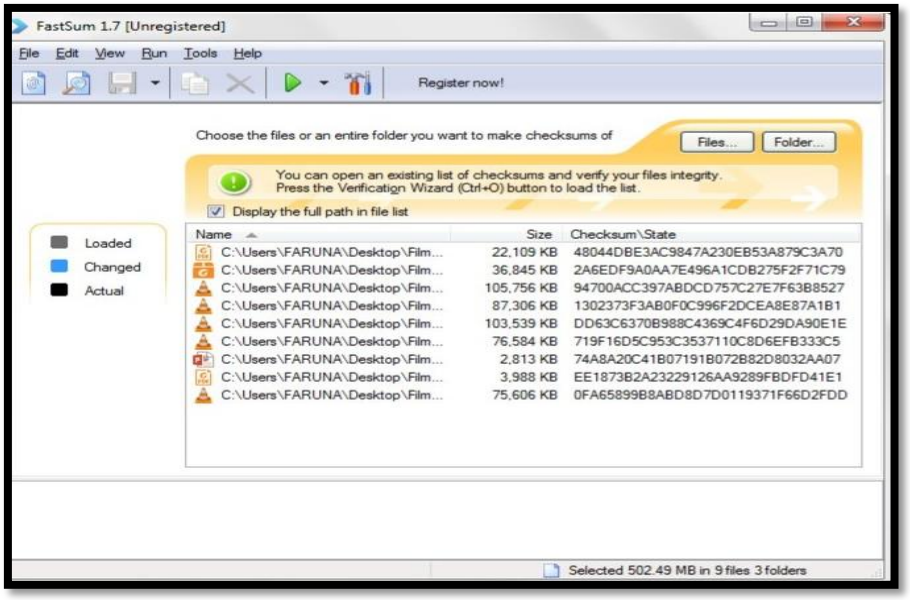

Fig.11. Integrity Test of Target Directory

To assess the performance of the system in respect of speed of synchronization, that is, time taken to complete a synchronization process, two sets of experiments were performed. The two experiments entail synchronization of data of sizes 100MB, 200MB, 300MB, 400MB, 500MB, 1000MB, 2000MB, 3000MB, 5000MB, and 10000MB. In the first experiment, each was a single video file. On the other hand, in the second experiment, each was the total size of multiple files. In both cases, the times taken to synchronize were compared with those by three other popular file synchronization tools: Synkron 1.6.2 [24], FreeFileSync 7.1 [25], and DirSync Pro 1.50 [26].

The tests were carried out with the aid of a computer system with the following specifications:

- OS Name: Microsoft Windows 7 Ultimate
- Version: 6.1.7601 Service Pack 1 Build 7601
- OS Manufacturer: Microsoft Corporation
- System Manufacturer: Hewlett-Packard
- System Model: HP 630 Notebook PC
- System Type: x64-based PC
- Processor: Intel(R) Pentium(R) CPU B960 @ 2.20 GHz, 2200 MHz
- RAM Size: 2.00GB, 1.85GB usable

Tables 1 and 2 show the synchronization time (in seconds) taken by the four tools for different files of one file type and different types respectively. In both sets of experiments, FileSyncer was found to be more efficient in terms of time taken to complete a synchronization operation. The efficiency level increases with corresponding increase in file size.

IX. Conclusion

In this study we presented a FileSyncer, a rapid and efficient file synchronization system which provides functionality for manual update selection and mechanism to revert a synchronization process back to the last previous state. The system was demonstrated to effectively synchronize between media. When compared with some popular open-source file synchronization systems, FileSyncer synchronized significantly faster. Our future plan is to release FileSyncer to the open source community for further development

Table 1. Comparison of the Various Synchronization Times for the Different File Synchronizers for One File Type

File Size(MB)	Synkron	FreeFileSync	DirSync	FileSyncer
100	3	3	3	1
200	5	5	6	4
300	13	11	13	10
400	19	17	21	16
500	26	25	25	23
1000	70	69	65	53
2000	128	146	125	108
3000	158	166	160	150
5000	270	276	268	260
10000	725	788	721	644

Table 2. Comparison of the Various Synchronization Times for the Different File Synchronizers for Different File Types

File Size(MB)	Synkron	FreeFileSync	DirSync	FileSyncer
100	12	14	15	12
200	30	32	31	31
300	40	42	40	40
400	53	54	53	52
500	63	62	64	61
1000	170	200	187	153
2000	266	289	290	241
3000	365	381	395	331
5000	780	790	798	760
10000	1589	1660	1646	1540

REFERENCES

[1] B. C. Pierce, and J. Vouillon, "What's in Unison? A formal specification and reference implementation of a file synchronizer," 2004. Retrieved June 17th, 2015 from http://repository.upenn.edu/cgi/viewcontent.cgi?article=1045&context=cis_reports

[2] N. Ramsey, and E. Csirmaz, "An algebraic approach to file synchronization," *ACM SIGSOFT Software Engineering Notes*, vol. 26(5), pp. 175-185, 2001.

[3] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall, "MetaSync: File synchronization across multiple untrusted storage services," 2015. Retrieved July 29th, 2015 from ftp://trout.cs.washington.edu/tr/2014/UW-CSE-14-05-02.PDF

[4] C. Liang, L. Hu, Z. Lei, and J. Wang, "SyncCS: a cloud storage based file synchronization approach," *Journal of Software*, vol. 9(7), pp. 1679-1686, 2014.

[5] B. Muruganantham, and T. K. Pandey, "Replica synchronization in distributed File system using asynchronous replication," *SSRG International Journal of Computer Science and Engineering*, vol. 2, pp. 12-17, 2015.

[6] M. S. Seemadevi, S. Y. Ramesh, and P. B. Dhainje, "Adaptive replica synchronization for distributed file systems," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 5, pp. 1368-1371, 2015.

[7] M. J. Vini, R. Nallathamby, and C. R. Robin, "A Novel Approach for Replica Synchronization in Hadoop Distributed File Systems," *Procedia Computer Science*, vol. 50, pp. 590-595, 2015.

[8] A. Lareida, T. Bocek, S. Golaszewski, C. Luthold, and M. Weber, "Box2Box - A P2P-based file-sharing and synchronization application," *In Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference*, pp. 1-2, September 2013.

[9] J. Lindblom, M. Huang, J. Burke, and L. Zhang, "File Sync/NDN: Peer-to-peer fileSync over Named Data Networking. NDN, Technical Report (NDN-0012)," 2013. Retrieved July 29th, 2015 from http://named-data.net/wp-content/uploads/TRFilesync.pdf.

[10] H. Zhang, C. Yeo, and K. Ramchandran, "VSYNC — a novel video file synchronization protocol," *In proceedings of the 16th ACM international conference on multimedia*, pp. 757-760, October 2008.

[11] Balasubramaniam, S., & Pierce, B. (1998). What is a File Synchronizer? In Proc. of the ACM/IEEE MOBICOM'98 Conference, pages 98–108, October 1998.

[12] Aspera, "Aspera sync. Scalable, multidirectional synchronization of big data – over distance," Retrieved August 10th, 2015 from http://asperasoft.com/fileadmin/media/Asperasoft.com/Resources/White_Papers/Sync_Asper aWP.pdf

[13] TGRMN Software, "FAQ and Knowledge Base," Retrieved from http://www.tgrmn.com/web/kb/item34.htm

[14] S. Khanna, K. Kunal, and B. C. Pierce, "A formal investigation of diff3," *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 485-496, December 2007, Springer Berlin Heidelberg.

[15] D. Gupta, and K. Sagar, "Remote file synchronization single-round algorithms," *International Journal of Computer Applications*, vol. 4(1), pp. 32-36, 2010.

[16] T. Suel, and N. Memon, "Algorithm for delta compression and remote file synchronization," 2002. Retrieved March 16th, 2015 from http://cis.poly.edu/suel/papers/delta.pdf

[17] T. Suel, P. Noel, and D. Trendafilov, "Improved file synchronization techniques for maintaining large replicated collections over slow networks," *Proceedings 20th International Conference on Data Engineering*, pp. 153–164, 2004, IEEE.

[18] A. Tridgell, and P. Mackerras, "The Rsync algorithm, "*Joint Computer Science Technical Report Series*, pp. 1-6, 1996. Retrieved July 29th, 2015 from https://cs.anu.edu.au/techreports/1996/TR-CS-96-05.pdf

[19] U. Irmak, S. Mihaylov, and T. Suel, "Improved single-round protocols for remote file synchronization," *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 3, pp. 1665-1676, March 2005.

[20] G. Shial, and S. Majhi, "Techniques for file synchronization: a survey," *Journal of Global Research in Computer Science*, vol. 5(11), pp. 1-4, 2014.

[21] H. Yan, U. Irmak, and T. Suel, "Algorithms for low-latency remote file synchronization," *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, April 2008.

[22] A. Dennis, B. H. Wixom, and R. M. Roth, "*System Analysis and Design*, 5th ed., Danvers, MA: John Wiley and Sons, 2012.

[23] FastSum, "Frequently Asked Questions: Basic Concepts. What is MD5 hash?," 2011. Retrieved October 16, 2015 from www.fastsum.com/support/md5-checksum-utility-faq/md5-checksum.php

[24] Synkron, "Folder Synchronization," 2011. Retrieved October 17th, 2015 from www.Synkron.sourceforge.net/

[25] FreeFileSync, "About FreeFileSync," 2015. Retrieved October 17th, 2015 from www.freefilesync.org/

[26] DirSync Pro, "What is it?," 2015. Retrieved October 17th, 2015 from www.disyncpro.org/

Authors' Profiles

Oluwafemi Osho is currently a lecturer in the Department of Cyber Security Science, Federal University of Technology, Minna, Nigeria. He holds a B.Tech. degree in Mathematics/Computer Science and an M.Tech. degree in Mathematics. Before joining the institution, he served as Head of the IT Department of one of the leading mortgage banks in Nigeria. His current research interests include cybersecurity, mobile security, and security analysis. He is a Certified Ethical Hacker (CEH).

Anthony Ugbede Faruna holds a B.Tech degree in Computer Science (Cyber Security).

How to cite this paper: Oluwafemi Osho, Anthony Ugbede Faruna,"FileSyncer: Design, Implementation, and Performance Evaluation", International Journal of Computer Network and Information Security(IJCNIS), Vol.8, No.11, pp.32-43, 2016.DOI: 10.5815/ijcnis.2016.11.04

