Computing and Informatics, Vol. 34, 2015, 1357–1373

INTRODUCING VERSIONING-BASED SOFTWARE ONLINE UPGRADE FRAMEWORK OVER A PEER-TO-PEER NETWORK

Kun Ma

Shandong Provincial Key Laboratory of Network Based Intelligent Computing
University of Jinan
South Xinzhuang 336
250022 Jinan, China
e-mail: ise_mak@ujn.edu.cn

Ajith Abraham

Machine Intelligence Research Labs
Scientific Network for Innovation and Research Excellence
P. O. Box 2259
98071 Auburn, USA
&
IT For Innovations
VSB – Technical University of Ostrava
70121 Ostrava, Poruba, Czech Republic
e-mail: ajith.abraham@ieee.org

Abstract. Speed and trend in the software evolution demand the more flexible and available software. For the large-scale software, the availability and automation of current software online upgrade approach is not ideal, which brings the managers much work to maintain all the versions of software. In this paper, we introduce the versioning-based software online upgrade framework (VSOUF) over BitTorrent-like Peer-to-Peer (P2P) Network. The distributed hash table (DHT) layer and version control (VC) layer is the core of this framework. Software clients can carry out initializing and upgrading by the atomic operations of a version control. Borrowing from P2P, we weave the distributed hash table (DHT) algorithm to speed up the download rate of a version control so that this framework will not crash under the

failure of a single node. Finally, experiments have showed the high performance and availability of the proposed framework.

Keywords: Software online upgrade, version control, distributed hash table, peer-to-peer network

1 INTRODUCTION

Many large-scale softwares, especially Cloud applications, Web 2.0 applications and massively multiplayer online (MMO) games, are expected to provide continuous service to their users without interruption or suspension of the service [1, 2, 3]. To achieve the continuous service, it must be possible to upgrade such a system by replacing individual software components with new revisions, while the system continues to operate. In this situation, the challenge is how to reduce the human intervention and how to manage all the repository revisions of software automatically.

Peer-to-peer (P2P) systems, as one of the most popular Internet applications, amaze even the experienced observers [4, 5]. Some researchers explore the unique strength of P2P in high-speed networks, identify performance bottlenecks, and quantify the special needs in the new scenario [6]. At the same time, version control is the management of multiple revisions. It is most commonly used to manage ongoing development of digital documents like source code, electronic models and other critical information that may be worked on by a team of people [7]. There are many previous studies on each of them. However, the theoretical benefits of the P2P paradigm in the Internet have been widely reported, it remains rare in a potential integration with the current software online upgrade approach.

In this paper, we have proposed a new Versioning-based Software Online Upgrade Framework (VSOUF) over BitTorrent-like P2P Network, in which this problem is resolved gracefully. We explore the potential integration of P2P paradigm and copy-modify-merge version control methods to solve the problems of current software online upgrades. In addition, the experimental result of this approach illustrates the optimization of fetching time and downloaded bytes of software online upgrades. The analysis of service disruption is included to show that the framework will not crush in the failure of a single node. Although the theory we present is similar to distributed version control systems (e.g. Git [8], DVCS [9]), the difference is that our version control approach is inspired by the peers in the swarms over P2P networks.

The contributions of this paper are multiple. First, it presents a design and implementation of versioning-based software online upgrade framework (VSOUF). Second, it is a new integration that the software online upgrade could benefit from both P2P networks and version control techniques. Third, it quantifies the fetching, rollback time and downloaded bytes to illustrate the performance of the proposed

framework. VSOUF is advantageous because it enables P2P distributed hash table (DHT) to accelerate the upgrade process. On the other hand, the upgrade will not crush in the failure of some nodes. This delivers the high availability of this framework.

The remainder of the paper is organized as follows. The related work is discussed in Section 2. Section 3 shows the goal statement of the ideal online upgrade framework. In Section 4, the architecture of our proposed VSOUF is investigated. The distributed hash table (DHT) layer and version control (VC) layer of this framework is discussed in detail. In Section 5, we present the discrete software online upgrade approach and its algorithms using the copy-modify-merge models. In Section 6, we analyze and demonstrate that the transfer optimization in VSOUF will reduce the fetch time and the downloaded bytes dramatically. Conclusions are provided in the last section.

2 RELATED WORK

Currently, there are at least three sorts of approaches to software online upgrade: package-based, hash-based and versioning-based software upgrades.

Package-based software upgrade is the main release management of Free and Open Source Software (FOSS) [10]. FOSS distributions are being made of thousands of components evolving rapidly without centralized coordination [11]. They define the granularity at which components are managed (installed, removed, upgraded to a newer version, etc.) using package manager applications, such as APT and YUM. Furthermore, this way affords an anarchic array of dependency modalities between the adopted packages. However, there are also shortcomings, especially in dealing with failures and rollback of upgrades.

Hash-based software upgrade identifies the new version by the MD5 or SHA1 of the software files. It indicates that the file of software needs an upgrade when and only when the MD5 or SHA1 of the local file is inconsistent with the remote file. Although it is common in the upgrade of online games, this approach forces the upgrade server to maintain the file hash list dynamically. It will cause the bottleneck once the files on an upgrade server are used in massive quantities.

Versioning-based software upgrade is a new approach to benefit from the file version control system (VCS). This approach considers that the software is composed of groups of files. Although there are lots of mature open-source version control systems, versioned metadata can consume as much space as versioned data [12]. Moreover, conventional version control systems do not efficiently support fault tolerance and concurrency control.

The main difficulties related to the management of versioning-based software upgrades depend on the management of revision repositories. In this respect, proposals like [13] represent a first step toward the rollback management. They exploit re-creation of removing packages on-the-fly, so that they can be re-installed to undo the upgrade. However, such approach can still need the manual intervention to main-

tain all the revisions. Moreover, the software cannot revert to any previous revision as it wants. Cicchetti shows how to apply model driven techniques to describe and manage software upgrades of FOSS distributions [14]. This approach represents an important advance with respect to the rollback of failed or unwanted upgrades. The above methods of software upgrades do not discuss the acceleration optimization in large-scale and heavy upgrades. Some open-source Linux package-management utility (e.g. DebTorrent [15] and apt-p2p [16]) has started implementing a P2P solution to a package distribution. Using a BitTorrent-like system, these voluntary mirrors could simply join the swarm of peers for the archive. However, these approaches are dependent on the Linux operating system. And the downloaded bytes of each upgrade is relatively high. Moreover, these upgrade tools need to maintain all the delta upgrade contents of packages.

Compared with the current software online upgrade approaches, our VSOUF solution has the following superior features. First, VSOUF support infinite rollback with the help of version control, which does not need to maintain the incremental upgrade data manually. VSOUF delegates all the software versions to version control systems automatically. For each upgrade, only the increment is downloaded. In addition, VSOUF works high availability environments by means of peers in the swarm. It is more reliable as central dependency is eliminated. Failure of one peer does not affect the functioning of other peers. In case of other solutions, if the upgrade server goes down the whole upgrade gets affected. Finally, VSOUF can accelerate the network traffic and increase the download speed of each upgrade.

3 GOAL STATEMENTS

Online upgrade becomes more of a strategy when the system encourages or automates this procedure directly. It tries to lower the barrier for all clients to move to the latest version. First, we propose the goal statements to improve the dependability of online upgrades by combining them with the industrial standards [17]. A reliable upgrade can also be approached from the four viewpoints: automation, atomicity, rollback and availability.

Automation of the upgrade: the whole process of software upgrade and management of the software repository is aimed to minimize the human intervention.

Atomicity of the upgrade: at any time, old version of an object is not allowed to service the client's requests once control has been turned over to the new version. Moreover, the end-to-end upgrade must be an atomic operation.

Rollback of the upgrade: if the new version of the object does not perform as expected, it must be possible to revert to the old version of the object.

Availability of the upgrade: when an upgrade size reaches the limitations of bandwidth, a mechanism is needed to allow users to upgrade software in a managed way.

4 ARCHITECTURE

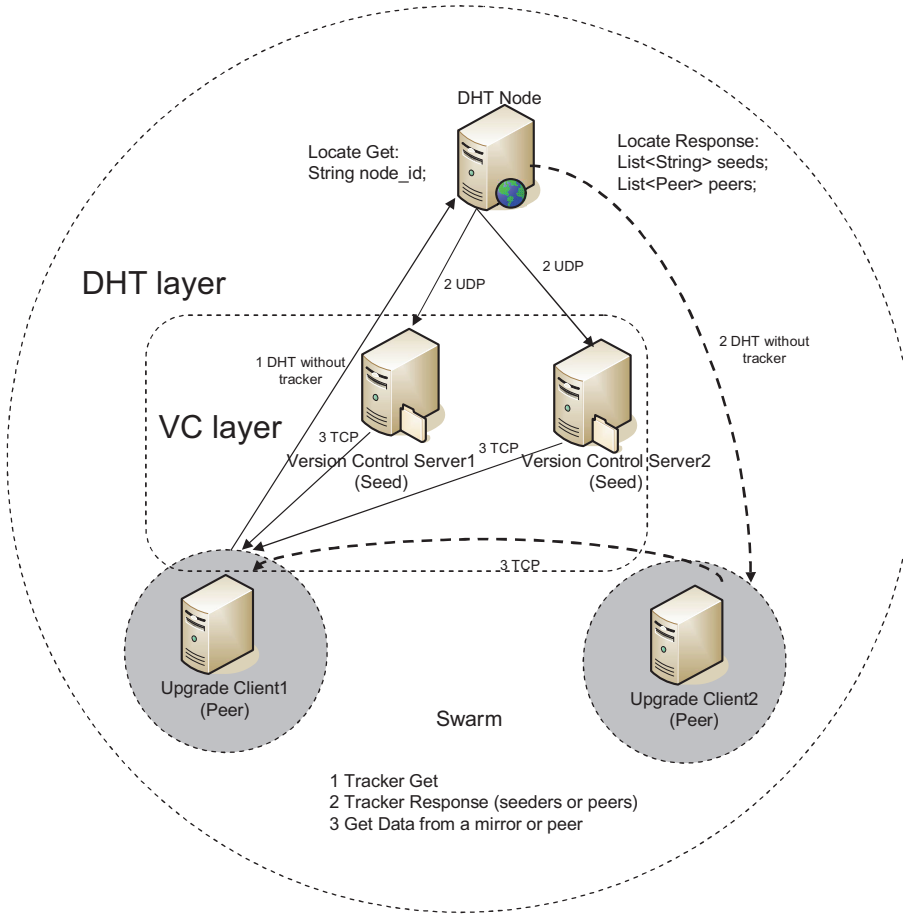

Figure 1. The architecture and software online upgrade process of VSOUF

There are several components in the scenario of software online upgrades. The first is upgrade client, which is the deployed software that needs updating frequently. VSOUF is designed to keep the clients latest, eventually through an incremental upgrade, to reduce the heavy workload. The second is upgrade server, which provides the repository of software to store all the historical versions. With the help of P2P acceleration, each upgrade client can download the data from several upgrade servers (seeds) and other upgrade clients (peers) in the swarm. The last is upgrade manager, that is instead of a manual maintenance of the revisions increments. In our VSOUF solution, we delegate the management of revisions to version control

systems automatically. For each upgrade, what the managers really do is committing the changes of software to the repository. At the same time, the upgrade clients in the swarm will detect the available updates.

The key point of VSOUF is weaving the file version control into ordinary peers instead of centralizing them in a single server or some specific ones. As shown in Figure 1, each peer acts as both the client and server respectively. Therefore, the load of the centralized version control server is balanced into peers in the swarms and single points of failure are avoided. Each file in the repository is managed by an appointed peer, which takes charge of the version control and information recording of the file.

VSOUF consists of two layers, distributed hash table (DHT) layer and version control (VC) layer. VC layer, inside DHT layer, stores detailed information on each file. Every upgrading operation on a file is logged. Whenever read/write operations on a file are initialized, the VC layer of client resorts to DHT layer to find out the corresponding repository seed and software online upgrade client peer (hereinafter abbreviated to upgrade client) of the requested file and retrieves its information.

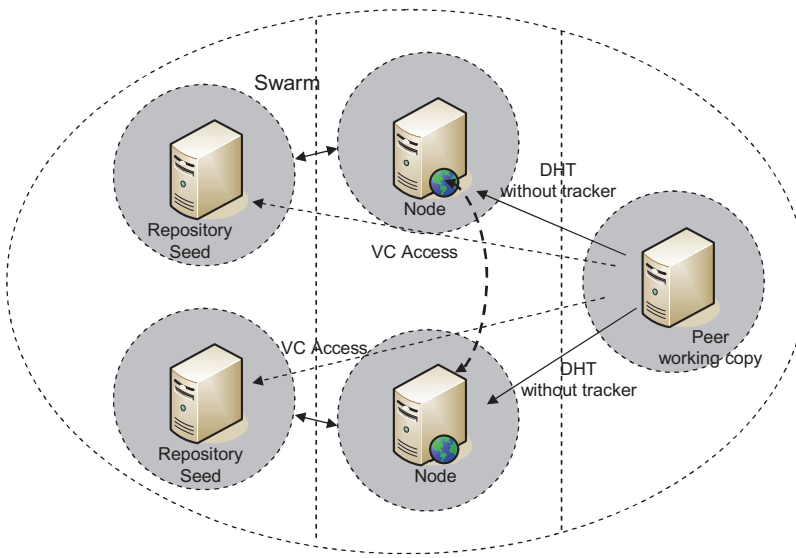

Figure 2. The distributed hash table layer of VSOUF

The main task of DHT layer is to locate the repository seed of requested files. We adopt the BitTorrent [18] Protocol in DHT layer to achieve this goal. As the improvement of BitTorrent, DHT is used to store peer contact information for tracker less torrents [18]. In effect, each node becomes also a tracker. DHT layer is composed of nodes and stores the location of peers (see Figure 2). There are some terms to explain the architecture. A peer is a client/server listening on a TCP port that implements the BitTorrent protocol; a node is a client/server listening on a UDP

port implementing the DHT protocol. It is used to find peers and seeds in the swarm; a seed is used to refer to a peer who has 100% of all the files of revisions. Upgrade clients include a DHT node, which is used to contact other nodes in the DHT to get the location of peers to download from using the BitTorrent protocol.

VC layer, which is built inside DHT layer, encapsulates the atomic version control operations and provides similar interfaces to normal file systems, such as checkout, update and ignore. On one end, there is a VC repository seed that holds all the versioned files of the software. On the other end, there is the upgrade client, which manages local reflections of portions of that versioned data (called working copy). We apply this version control approach in the scenario of the software online upgrade. It resorts to DHT layer to look up the repository seed and the upgrade client of a given file and takes charge of checkout and the upgrade of managed files. The repository stores all the revisions with a specified data structure. All updating operations on file are logged.

We have made further improvement to employ the P2P network to speed up the version control solution of the software online upgrade. The whole process of versioning-based software online upgrade is discussed in detail (see Figure 1). First of all, the upgrade client requests the DHT closer node to get the other upgrade clients and seeds of the repository by the unique node id. In addition, the DHT node returns the list of seeds and upgrade clients. Finally, the upgrade client updates to the latest revision when the local and remote revision number is inconsistent. To avoid overloading the traffic of networks, VC layer of each peer caches the files. Not only the contents but also the versions are cached. This mechanism, that client peers also share its revision, assures the acceleration of software online upgrade. In this paper, we employ a copy-modify-merge model [19] as an alternative to the version control. This merge-based mechanism is borrowed from version control techniques to achieve the data consistence.

5 SOFTWARE ONLINE UPGRADE APPROACH

The scenario, timeline and implementation details are given in this section. First, we illustrate a usual scenario of the software online upgrade, in which an ideal copy-modify-merge VC is adopted.

5.1 Timeline of Software Online Upgrade Approach

The timeline of an example is shown in Figure 3. In this scenario, the upgrade manager first checks out the repository. Second, the developer will rectify the software. Once there is a new release of the software, the developers will commit all the changes to the repository with a log message. Then, the upgrade client will create a personal working copy – a snapshot of the repository to formulate the initial revision of the software. Also the ignore operation is executed to neglect the unique personal file (generally it is the config file), which will not participate in the next

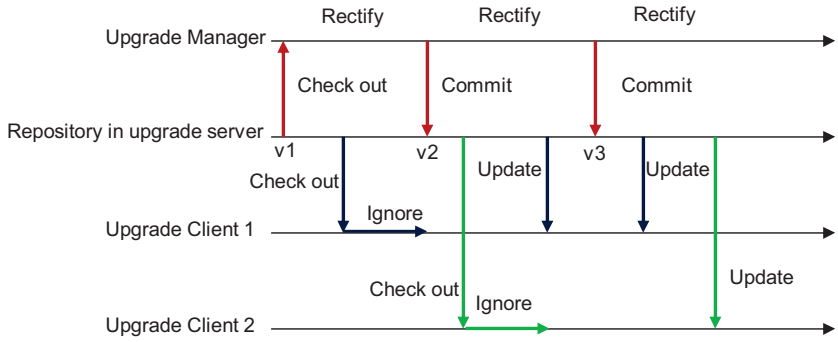

Figure 3. The timeline of software online upgrades

upgrade. The upgrade clients will detect the changes of repository all the way. After the developers commit the next changes to the repository, the upgrade client can update to HEAD to upgrade the working copy to formulate a new software revision. As the client will not modify the working copy besides the unique ignored personal file, the conflict in the update operation will not appear.

5.2 Copy-Modify-Merge Software Online Upgrade Approach

If we want to formalize VSOUF, we must begin by defining the copy-modify-merge approach over P2P networks.

- **File** V_i: a version of a file in the working copy, where i is the revision number, e.g. V_2.
- **Base file** V_B, with B for BASE: the last file to be checked out or update prior to any modification.
- **Head file** V_H, with H for HEAD: the latest (or most recent) file in the working copy. Note that the head file is the same for all clients. In contrast, the base file is bound to the working copy, and may differ from user to user.
- **Deltas against files** Δ, is a patch file from V_i to V_j, $\Delta := V_j - V_i$, where $i < j$.
- **Merging Deltas**, applying deltas Δ against the current file V_i will generate a new file V_j, denoted as $V_j = V_i + \Delta$.
- **Working copy**, denoted as 2-tuple $workingCopy := (\{V_i | V_i \text{ is a file}\}, latestFileSystem)$, where $latestFileSystem$ is the filesystem tree.
- The file in the working copy is denoted as $V = V_B + \Delta_1 + \Delta_2 + \ldots + \Delta_n$, where n is times of deltas files Δ. Generally, $V_H = V_B + \Delta_1 + \Delta_2 + \ldots + \Delta_m$, where m is times of deltas files Δ from V_B to V_H.
- In order to describe the algorithm, the projection operator is introduced to present the i_th projection of n-ary sequence, denoted as $i\_th(a_1, a_2, \ldots, a_n) = a_i$, e.g. $2\_th(workingCopy) = latestFileSystem$.

In order to calculate the difference between two subsequent versions of a file, we need to know which modifications have taken place. The identifying these modifications requires an analysis of the old and the new versions of the file. The copy-modify-merge version control model lives and dies on its merging algorithms. We provide only one such algorithm: a three-way differencing algorithm [20] that is smart enough to handle data at a granularity of a single line of text. It allows supplementing its content merge processing with external differencing utilities.

We can begin to define operations that somehow change the repository. In particular, we consider the following file atomic operations on repositories: *locate*, *checkout*, *commit*, *update* and *ignore*.

- The *locate* operation is used for the upgrade client to find the peers and seeds in the swarms.
- The atomic operation *checkout* means checking out a repository to create a working copy of it on your local machine. This working copy is $workingCopy := (\{V_{Bi}|V_{Bi} \text{ is a file}\}, latestFileSystem)$. This copy contains the HEAD (latest revision) of the repository that constitutes the initial software. Note that we fetch the downloaded file from lots of repository seeds at the same time.
- After checking out the working copy of the repository seed, the atomic *commit* operation sends all of changes of software to the repository to formulate a new revision. Then, the repository seed will synchronize to the other repository peer in the swarms. For each file model V_i in the working copy *workingCopy*, the base model V_B changed into V_H, denoted as $(V_i' = V_{iB}' + \varnothing) \leftarrow (V_i = V_{iB} + \Delta_1 + \Delta_2 + \ldots + \Delta_n)$, where $V_{iB}' = V_{iB} + \Delta_1 + \Delta_2 + \ldots + \Delta_n$.
- To bring the working copy up to date, the atomic *update* operation incorporates the changes of the repository into its working copy, as well as any others, that have been committed since the upgrade client checked it out. This working copy is $workingCopy := (\{V_i|V_i \text{ is a file}\}, latestFileSystem)$, where $V_i = V_B + \Delta_1 + \ldots + \Delta_n$. In the scenario of the software online upgrade, the local files, except the personal files, in the working copy will not be changed, and the file should eventually be updated in order to make it current with the latest revision of the repository. Note that there are two kinds of update. The former is the update to *HEAD*, which means updating the working copy to the latest revision, and the latter is the update to *VERSION*, which is a useful approach to revert the software into the previous revision. The revert operation depends on deltas against files.
- The *ignore* operation is the mechanisms to use file patterns (strings of literal and special wildcard characters used to match against filenames) to determine which files to be ignored. The file ignored will keep unmodified the next upgrade.

The algorithms *initialize* and *upgrade* is the core function of VSOUF. Most of the time, the software will initialize by executing Algorithm 1. When there is a new release of the software, the repository server sends the changes to your working copy via *upgrade* operation (see Algorithm 2).

Algorithm 1: Initialize

Input:
Node ID *nodeId*;
Regular expression list of ignore *ignorePatternList*;
Output:
Working copy *workingCopy*;
1: *request.node_id* = *nodeId*;
2: *response* = locate(*request*);
3: *workingCopy* = checkout(*response.seeds*);
4: ignore(2_*th*(*workingCopy*), *ignorePatternList*);
5: **return** *workingCopy*;

Algorithm 2: Upgrade

Input:
Node ID *nodeId*;
Regular expression list of ignore *ignorePatternList*;
Working copy *workingCopy*;
Output:
Working copy *workingCopy*;
1: *request.node_id* = *nodeId*;
2: *response* = locate(*request*);
3: ignore(2_*th*(*workingCopy*), *ignorePatternList*);
4: *workingCopy* = update(*response.seeds*, *response.peers*);
5: **return** *workingCopy*;

6 EXPERIMENTAL RESULT

The experiments were conducted on a Xeon(R) Core(TM) CPU E7-4807 1.86 GHz 6 Core server with 8 GB RAM, a 1 GB Realtek 8169 NIC in 100 M LAN. The systems were configured with a LINUX CENTOS 6.3, with kernel version 2.6.32-279. We have developed the evaluating prototype using VSOUF on this server. All the experiments are performed on a 16-peers cluster.

6.1 Fetching Time

If the upgrade client does not own the right file of revision, it has to fetch the whole file from the repository seeds or peers in the swarm. The experiment was performed to evaluate the cost of fetching different sizes of file in multiple peer swarms. Supposed, there is a file to be updated from version 1 to 2. Figure 4 shows the result of fetching 1 M, 50 M, 200 M and 500 M bytes of files from VSOUF with 0 peers, 4 peers and 16 peers. The results indicate that the acceleration effect is obvious when the number of peers becomes greater and the files get larger. It is easily explained theoretically that P2P network plays an important role in large file

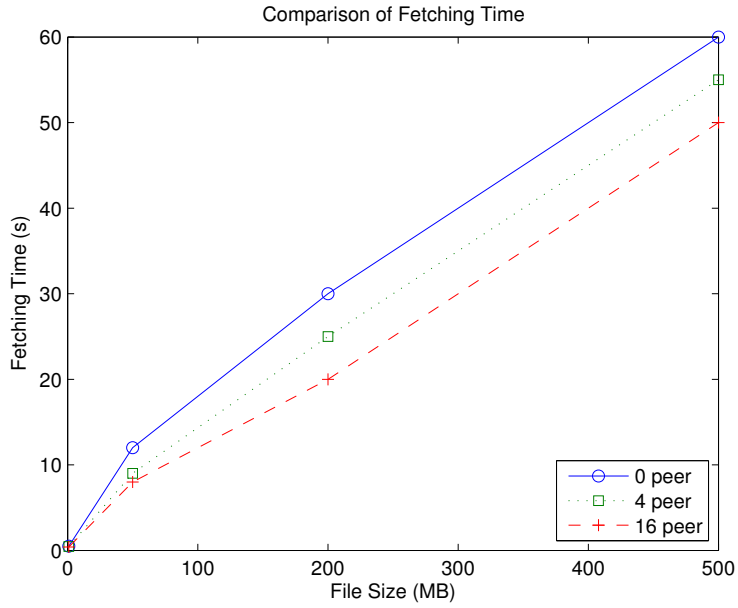

Figure 4. Comparison of fetching time upgraded from version 1 to 2

transfer. This experiment demonstrates that the transfer optimization in VSOUF reduces the fetch time dramatically.

Next, we issue an upgrade from the same original revision to the same new revision to measure the fetching time using different software online upgrade approaches. We use threads to simulate the online upgrade clients. Figure 5 shows the fetching time of each upgrade with package-based, hash-based, versioning-based and our VSOUF solutions. The fetching time of VSOUF solution remains constant with different threads. When the number of threads exceeds 200, the fetching time except our VSOUF solution becomes larger since this is caused by the bottleneck of a single upgrade server. This result also illustrates that P2P acceleration of our VSOUF solution is obvious especially under high concurrency as well.

6.2 Rollback Time

Besides the fetching time, we measure the update time for rolling back of the update. First of all, we update an application from its initial version to its last version and then roll back to the first version. Figure 6 shows the update time of our measured file from version 1 to 3. The update time includes the whole process to apply a patch. It includes loading the patch into memory, download files from the repository or peers in the swarm. As shown in Figure 6, the time to roll back is very short due to deltas against files. Though not exactly, the update time is also related the patch

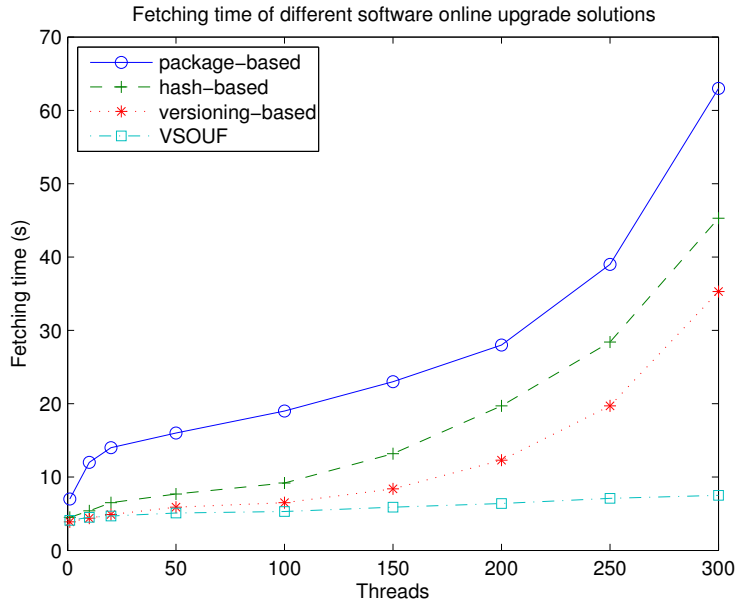

Figure 5. Comparison of fetching time of different solutions

size which roughly reflects the number of changes.

Approach	Rollback
package-based	limited support using downgrading or repackage
hash-based	not clear
versioning-based	infinite rollback using version control
VSOUF	infinite rollback using version control

Table 1. Rollback support

Next, we compare the rollback support of different solutions. As depicted in Table 1, the package-based solution provides the rollback using downgrading or repackage operation. Generally, it needs many workloads to define the increment of each upgrade. It is not clear that hash-based solution can support rollback. Versioning-based and our VSOUF solution support infinite rollback capabilities with the help of remote repository. This needs no extra data from the server. However, our VSOUF has more superior features than the versioning-based approach, such as high availability and P2P download acceleration.

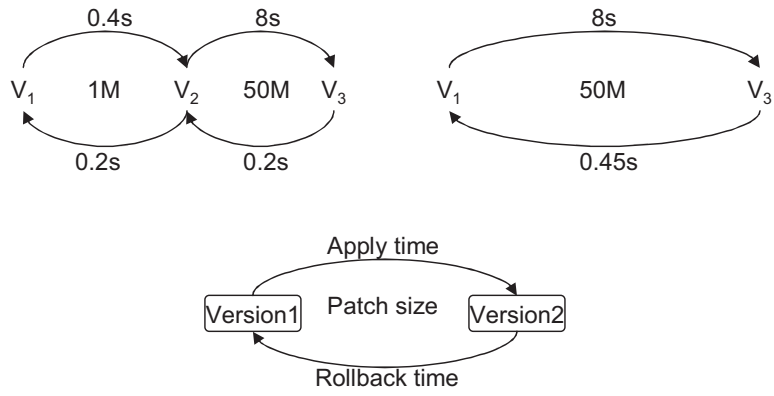

Figure 6. Update time and rollback time

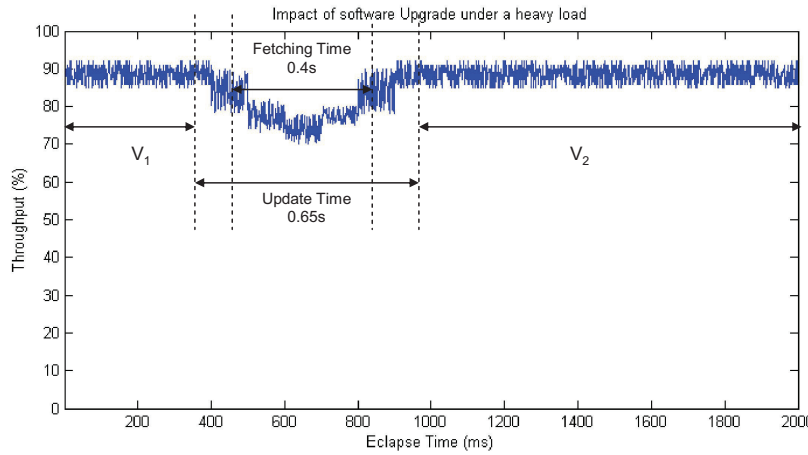

Figure 7. Impact of software upgrade under a heavy load

6.3 Service Disruptions

To measure the impact of software upgrade on running services, we use Apache Benchmark [21] to issue 20 000 requests for a single 1.0 MB file with 16 peers and collect the throughput of Apache httpd server when an update from version 1 to 2 is in progress. The command is "*ab -n 20000 -c 500 http://localhost/test.png*". Figure 7 depicts the curve of throughput during the process of updating. There is only a modest amount degradation (about $40\% = 1 - 0.4/0.65$) during the update. Besides, even under a heavy system load, the increase in update time is still modest (from 0.4 s in Figure 4 to 0.65 s in Figure 7). Further, the time to evolve Apache httpd is still very short, even under a heavy load.

Approach	Disruption	Result
package-based	server crashes	failure
hash-based	server crashes	failure
versioning-based	server crashes	failure
VSOUF	single server crashes	success
VSOUF	all the seeds crash	might success (got from the peers in the swarm)

Table 2. Service availability of different solutions

Next, we analyze the availability when some service providers crash. As depicted in Table 2, only our VSOUF solution will be available by the seeds and peers in the swarm.

6.4 Downloaded Bytes

Approach	$0 \to 1$	$1 \to 2$	$2 \to 3$	$3 \to 4$	$4 \to 5$
package-based	5 190 k	8 305 k	55 158 k	1 245 k	123 735 k
hash-based	4 230 k	3 623 k	15 613 k	657 k	50 374 k
versioning-based	1 457 k	2 132 k	7 216 k	234 k	21 158 k
VSOUF with 4 peers	1 503 k	2 251 k	7 423 k	297 k	22 238 k

Table 3. Downloaded bytes of different solutions

We deploy the same software historical revisions (initial version is 0, and the next version is 1, 2, 3, 4, 5 respectively) using the package-based, hash-based, versioning-based and our VSOUF solution for further evaluation. Next, we measure the downloaded bytes to issue five upgrades (the change is $0 \to 1, 1 \to 2, 2 \to 3, 3 \to 4, 4 \to 5$) using different software online upgrade approaches. As depicted in Table 3, the downloaded bytes of versioning-based and our VSOUF solution are smaller than other solutions because these two solutions only download the incremental data of a changed file. And the downloaded data of VSOUF is just a little higher than the versioning-based solution. That is because VSOUF needs extra communications with the seeds and peers in the swarm. The additional bytes are considered negligible compared to the high availability and P2P download acceleration.

7 CONCLUSION

We have presented the design and evaluation of VSOUF, a new versioning-based software online upgrade framework over a peer-to-peer network in this paper. It provides upgrade services to users through a series of atomic version control operations, such as *locate*, *checkout*, *commit*, *update* and *ignore*. The primary goal of this paper is employing DHT algorithm to locate the repository peer of requested files and VC layer to manage the version information of files in the framework. As

a result of the considerate mechanism of peer joining and departuring, it still runs smoothly under single node failure scenarios. Improvements are still needed in the framework, but the current results have shown a good performance of VSOUF.

This framework achieved its goal statement introduced in Section 3 from the four viewpoints. First of all, the repository and working copy are managed by the version control framework. What to do before each upgrade is committing the new version working copy to the repository by the managers. And then, each client peer can upgrade its local working copy. This framework enables many operations to be entirely automated. Second, the atomic version control operations of this framework guarantee the atomicity of the upgrade. Third, we provide an update to *REVISION* operation to ensure the backtrack of software upgrades. Finally, thanks to integration with P2P network, this framework becomes more available when some peer nodes fail. The more peers in the swarm will guarantee the best download speeds.

Acknowledgment

This work was supported by the Doctoral Fund of University of Jinan (XBS1237), the Shandong Provincial Natural Science Foundation (ZR2014FQ029), the Shandong Provincial Key R & D Program (2015GGX106007), the Teaching Research Project of University of Jinan (J1344), the National Key Technology R & D Program (2012BAF12B07), and the Open Project Funding of Shandong Provincial Key Laboratory of Software Engineering (2015SE03).

REFERENCES

[1] Pan, Y.—Tang, Y.: LIGHT: Pay-As-You-Go Software Artifacts Management. Computing and Informatics, Vol. 32, 2013, No. 4, pp. 827–843.

[2] Ma, K.—Sun, R.—Abraham, A.: Toward a Module-Centralized and Aspect-Oriented Monitoring Framework in Clouds. Journal of Universal Computer Science, Vol. 19, 2013, No. 15, pp. 2241–2265.

[3] Ma, K.—Yang, B.: Multiple Wide Tables with Vertical Scalability in Multi-Tenant Sensor Cloud Systems. International Journal of Distributed Sensor Networks, Vol. 2014, 2014, Article ID 583686, pp. 1–10.

[4] Tang, Y.—Zhou, S.—Xu, J.: LIGHT: A Query-Efficient Yet Low-Maintenance Indexing Scheme over DHTs. IEEE Transactions on Knowledge and Data Engineering, Vol. 22, 2010, No. 1, pp. 59–75.

[5] Gupta, A.—Awasthi, L. K.: Peer-to-Peer Networks and Computation. Computing and Informatics, Vol. 30, 2011, No. 3, pp. 559–594.

[6] Chen, Z.—Zhao, Y.—Lin, C.—Wang, Q.: Accelerating Large-Scale Data Distribution in Booming Internet: Effectiveness, Bottlenecks and Practices. IEEE Transactions on Consumer Electronics, Vol. 55, 2009, No. 2, pp. 518–526.

[7] Ruparelia, N. B.: LIGHT: The History of Version Control. ACM Sigsoft Software Engineering Notes, Vol. 35, 2010, No. 1, pp. 5–9.

[8] Alwis, D. B.—Sillito, J.: Why Are Software Projects Moving from Centralized to Decentralized Version Control Systems? Proceedings of 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering (CHASE '09), Vancouver, May 2009, pp. 36–39.

[9] Satish, L.: Identifying the Dissimilarities Based on Working of Programs Among Versions in DVCS (Distributed Version Control Systems). International Journal of Computer Applications, Vol. 36, 2011, No. 6, pp. 25–29.

[10] Di Cosmo, R.—Zacchiroli, S.—Trezentos, P.: Package Upgrades in FOSS Distributions: Details and Challenges. Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades (HotSWUp '08), Nashville, October 2008, Article No. 7.

[11] Cosmo, R. D.—Ruscio, D. D.—Di Ruscio, D.—Pelliccione, P.—Pierantonio, A.—Zacchiroli, S.: Supporting Software Evolution in Component-Based FOSS Systems. Science of Computer Programming, Vol. 76, 2011, No. 12, pp. 1144–1160.

[12] Soules, C. A. N.—Goodson, G. R.—Strunk, J. D.—Ganger, G. R.: Metadata Efficiency in Versioning File Systems. Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03), San Francisco, March 2003, pp. 43–58.

[13] Trezentos, P.—Di Cosmo, R.—Lauriere, S.—Morgado, M.—Abecasis, J.—Mancinelli, F.—Oliveira, A.: New Generation of Linux Meta-Installers. Research Track of FOSDEM 2007, Brussels, February 2007, pp. 1–8.

[14] Cicchetti, A.—Di Ruscio, D.—Pelliccione, P.—Pierantonio, A.—Zacchiroli, S.: A Model Driven Approach to Upgrade Package-Based Software Systems. Communications in Computer and Information Science, Vol. 69, 2010, pp. 262–276.

[15] Debian Team, DebTorrent. Available on: `http://wiki.debian.org/DebTorrent`, 2009.

[16] Dale, C.—Liu, J.: apt-p2p: A Peer-to-Peer Distribution System for Software Package Releases and Updates. Proceedings of 28th Conference on Computer Communications (INFOCOM 2009), Rio de Janeiro, April 2009, pp. 864–872.

[17] Object Management Group: Online upgrades 1.0. Available on: `http://www.omg.org/spec/ONUP/1.0`, 2008.

[18] Loewenstern, A.: DHT Protocol. Available on: `http://www.bittorrent.org/beps/bep\_0005.html`, 2008.

[19] Rossini, A.—Rutle, A.—Lamo, Y.—Wolter, U.: A Formalisation of the Copy-Modify-Merge Approach to Version Control in MDE. The Journal of Logic and Algebraic Programming, Vol. 79, 2010, No. 7, pp. 636–658.

[20] Lindholm, T.: A Three-Way Merge for XML Documents. Proceedings of the 2004 ACM Symposium on Document Engineering (DocEng '04), Wisconsin, October 2004, pp. 1–10.

[21] Apache Software Foundation: Apache HTTP server benchmarking tool. Available on: `http://www.apache.org/`, 2012.

Kun Ma received his Ph.D. degree in computer software and theory from Shandong University, Jinan, Shandong, China, in 2011. He is Senior Lecturer in Provincial Key Laboratory for Network Based Intelligent Computing and School of Information Science and Engineering, University of Jinan, China. He has authored and coauthored over 30 research publications in peer-reviewed reputed journals and conference proceedings. He has served as the program committee member of various international conferences and as reviewer for various international journals. He is the Co-Editor-in-Chief of International Journal of Computer Information Systems and Industrial Management Applications (IJCISIM). He is the managing editor of Journal of Information Assurance and Security (JIAS) and Information Assurance and Security Letters (IASL). He is the editorial board member of International Journal of Intelligent Systems Design and Computing and Journal of Software. He is the Guest Editor of International Journal of Grid and Utility Computing. His research interests include model-driven engineering (MDE), data intensive computing, big data management, and multi-tenant techniques.

Ajith Abraham received his Ph.D. degree in computer science from Monash University, Melbourne, Australia. He is currently the Director of Machine Intelligence Research Labs (MIR Labs), Scientific Network for Innovation and Research Excellence, USA, which has members from more than 85 countries. He has a worldwide academic and industrial experience of over 23 years. He works in a multi-disciplinary environment involving machine intelligence, network security, various aspects of networks, e-commerce, Web intelligence, Web services, computational grids, data mining, and their applications to various real-world problems. He has numerous publications/citations (h-index 60) and has also given more than 60 plenary lectures and conference tutorials in the above mentioned areas. Since 2008, he is the Chair of IEEE Systems Man and Cybernetics Society Technical Committee on Soft Computing and a Distinguished Lecturer of IEEE Computer Society representing Europe (since 2011). He is a Senior Member of the IEEE, the Institution of Engineering and Technology (UK) and the Institution of Engineers Australia (Australia), etc. He is the founder of several IEEE sponsored conferences, which are now annual events.