ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Website: www.ijirset.com

Vol. 6, Issue 6, June 2017

Automated Update Mechanism using Streams and MD5 Algorithm designed for Remote Clients

Sohel S. Shaikh[1], Dr. V. K. Pachghare[2]

P.G. Student, Department of Computer Engineering, College of Engineering, Pune, Maharashtra, India[1]

Associate Professor, Department of Computer Engineering & IT, College of Engineering, Pune, Maharashtra, India[2]

ABSTRACT: At the present time, with evolving technologies there is a constant need to update the software applications to make them capable of producing efficient and optimal business solutions. However, the reflection of updates to applications is a tedious task. Propagating the updates online can reduce the expense and increase the convenience of the clients. This paper explores the existing approaches for updating the applications and identifies the pitfalls associated with them. The paper proposes an innovative technique for updating the clients in an easy, speedy and reliable manner.

KEYWORDS: Auto update strategy, MD5 algorithm, Streams, REST APIs

I. INTRODUCTION

Software updates are required for adapting the existing applications with capabilities to serve the evolving business requirements. The updates can be propagated online with ease. However, sending the updates over the wire has several vulnerabilities associated with it. It is susceptible to issues such as bad maintenance, complications in collocations, inconvenience at the time of reflecting the updates and high cost associated with maintenance.

This paper analyses several updating strategies and points out the associated highs and lows. These approaches satisfy the immediate requirements of the applications, however, there is no standard approach available auto updating the clients. This paper makes the use of REST architecture to communicate between the server and client for propagating the updates. It proposes an innovative, robust and reliable technique for updating the clients with ease.

II. THEORETICAL BACKGROUND

The existing system consists of a server-client architecture, wherein the updates are propagated by the server to every individual client. The server notifies the client of the available update. It then seeks remote access of the client to reflect the update. Along with the update a script is also sent. This script is executed once the update gets downloaded on the client. This script then makes the client system to reboot and reflect the update. This approach requires manual efforts to obtain the clients machine access remotely. Also accessing the client machine for reflecting the updates is not advisable.

This approach though seems to appear simple and quick to grasp. It is very time consuming and involves huge redundant manual labour. Moreover, this approach is still feasible for few clients, once the count of clients increases it results into increase in errors associated with it. The several risks identified across the traditional approach are well organized in the following section.

III. CHALLENGES ASSOCIATED WITH TRADITIONAL APPROACH

The challenges are listed below categories into five different sets for better understandability.

1. Vulnerability to human errors

In traditional server client application, there is no provision for reflecting a minor / major change at the server to the client side. They lacked a robust automated mechanism for updating client side algorithms or logic. The client had no mechanism for checking updates on the server. Hence it involved tedious, redundant manual task of obtaining control of

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Website: www.ijirset.com

Vol. 6, Issue 6, June 2017

every client, either remotely or by explicitly sending an update to the client and then instructing them to download the updated files. Further a shell script was pushed to client which would then execute by copying the downloaded files to lib folder and rebooting the client to reflect the updates. The very involvement of human made it susceptible to human errors.

2. *Huge memory consumption*

For propagating the updates all the update files are wrapped in a zip archive storing them on a physical disc at the server and then returned to the client. The forwarding of updates is also done in multiparts. On reception, client stores it on its local disk at a temporary location. After this it unzips the updates in another temporary location and send that data back to server for verification. Until this the server is also required to reside the zip archives for client to download in scenarios of failed verifications.

This requires an additional task at the server and client side for deletion of the zip archive files after upgradation. Let's assume if 1000 clients are requesting server for updates then server needs to store zip files for all these 1000 clients, which is definitely a cumbersome task

3. Maintenance of multiple updates simultaneously.

If there were more than one update available at the server, it would require the entire updating process to be repeated for each update. For every update, the server would have to create a zip archive and propagate it to the client. For every client, the server had to persist this huge archive files of all updates until every client suffices the update.

4. Difficult to backtrack to stable state

The update over the internet may get hit by miscommunication leading to improper update reflection at the clients. The very involvement of manual labor makes this approach more susceptible to human errors. Also rebooting of the client proposes another severe issue. If there is any discrepancy during this reboot it will not be logged by the system. Hence it will be impossible to backtrack the erroneous situation leading the system to stand in a non-functional state.

5. Susceptibility to security breaches

The jars are transported over the internet as plain files. Hence it makes it susceptible to several attacks such as interception of files over the wire, man-in-middle attack, etc. Interception of files over the wire keeps the important business logic at risk. Also, there is a threat that these files may get modified by the attackers and lead to malfunctioning of the system.

All these drawbacks led to an urgent need of an automated process for reliably upgrading the clients with minimal manual interference.

IV. PROPOSED METHODOLOGY

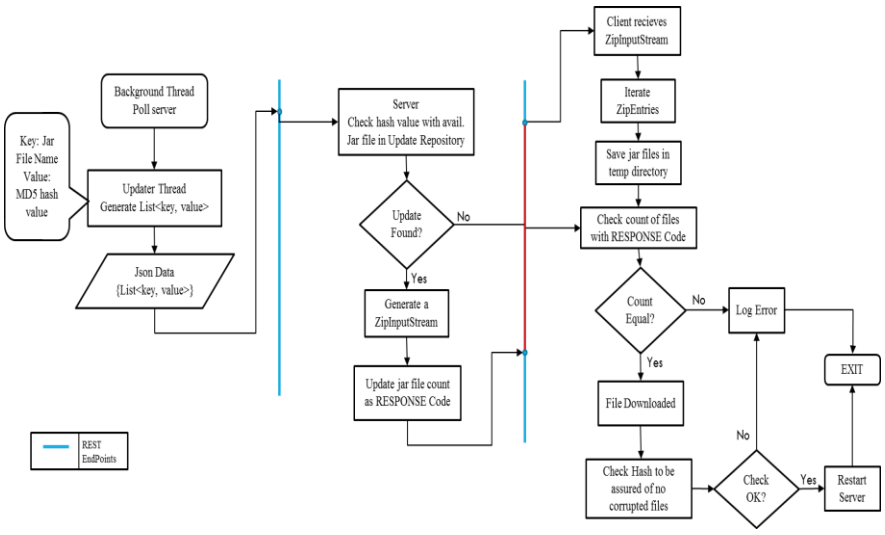

Figure 1 Flow Diagram of Proposed System

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Website: www.ijirset.com

Vol. 6, Issue 6, June 2017

The above diagram presents the flow diagram of the proposed model. The proposed technique relieves the task of a single server for sending the updates to every client. The client handles this task, a separate thread frequently polls the server to check for updates. The client is provided with an REST Endpoint to query the server about the updates.

The client hits the REST API by sending the List of <key value> pairs of the available jars. The server on being invoked by the client checks its update registry with the provided list. If inconsistency is found then it indicates the client about the available update. This system does not rely on the timestamp of the files as it malfunctions across different time zones and hence is found to be unreliable.

On being indicated about the updates the client fetches the updates and reflects it on its side without any human intervention. Hence, the proposed system is resistant to human errors.

V. IMPLEMENTATION

The algorithm developed is as presented below:

Algorithm: Auto Update Strategy

```
1.  Start Background Thread: Poll server to check if update available.

2.  Generate a list of <Key, Value> pair.

        1.  Key: name of jar file.

        2.  Value: MD5 hash value of the corresponding jar.

3.  Post this list as Json data to server.

4.  Server's REST endpoint: check the hash value of each jar file with available jar file in Update
    respository.

5.  If (Update found): Generate a ZipInputStream, put update jar file count as RESPONSE Code to client.
    else: Go to step 7

6.  Client Receives the ZipInputStream: iterate through ZipEntries saves jar files on the fly.

7.  Check count of downloaded files in temporary directory with RESPONSE Code received from server

        IF (UpdateJarFileCount=ResponseCode): File downloaded successfully.
        ELSE: Some Error occurred during file download. Log Error go to Step 10.
8.  Check HASH of jar file downloaded in updated directory with server, to check all files downloaded
    properly, they are not corrupted.

9.  IF (HASH Check successful): Copy updated jar to lib directory restart the service.

    ELSE: Go to Step 10
10. Exit
```

The above algorithm is broken into key steps and described below in detail:

A. Check for updates

The background thread continuously keeps polling the server by hitting the provided REST API. The updater thread creates a list of <Key, Value> pair in the JSON format, where key is the name of jar file and value is the MD5 hash of the corresponding jar. This JSON data is passed as request body while calling the REST API. This is all done at the client side.

B. Find information of the available update

On receiving the request, the server then checks the hash value of the jar files with available jar files in the update repository. If an update is found it sends a response to the client with the count of the files and prepares a ZipInputStream of the update files. The compress format of the files is selected as zip. It is well identified that the

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Website: www.ijirset.com

Vol. 6, Issue 6, June 2017

conventional compress format is rar, yet there is not an equivalent software performing in Linux, and zip implements this commonality. The server does not create any zip file in the local memory, but creates an input stream of it. This stream is then sent to the clients.

C. Download the update package using streams

The client receives the update package in the form ZipInputStream. The client then iterates through these zip entries and saves the jar files on the fly in a temporary directory. It then compares the count of the downloaded files in the temporary directory with the count mentioned in the server response. If the count matches then it proceeds to next step, else an exception is raised and the stream is fetched again.

D. Verify the update package

Once the client is assured about the complete package being downloaded, it then verifies if the package is unaltered and error free. It checks for the update package to be complete and usable. It is done by calculating the hash of the jars and cross checking it with the server. If the hash values match, the client is ready for updating its software.

E. 'Update

All the new versions of files needed to be updated are present in the update package. Once the client is assured of the authenticity of the downloaded package, it copies it into its lib directory. It then reboots the service to reflect the update.

VI. OBSERVATIONS & RESULTS

The proposed strategy was implemented using Java framework. The results obtained shows the following outcomes and significance of the proposed strategy.

A. Human Errors are eliminated

The proposed technique relieves the server of approaching every single client for reflecting the changes. The client itself polls for updates and discovers updates. The client is made self-reliant for downloading the updates as well as checking the validity and integrity of updates leading to least dependency on the server. It does not require any manual configuration, thus eliminating human errors completely.

B. Reduced Memory Consumption

Ideally, the approach would require server to create a zip archive of the update files and store it in its local memory. Then send this archive to the client in multipart response owing to the zip's size. These multipart would then get accumulated over the client in an archive. The client would store this archive in its memory, validate it with the server and then unzip it. Finally, after completion the server and client would their respective archives. This seems satisfactory with one client and one update at a time. However, for multiple clients and multiple updates it succumbs to failure and unlimited memory consumption. Clearly, for large responses the above creates an O(n) demand on the JVM heap memory: the more data is served, the closer the server gets to an out-of-memory condition. This does not only put a hard limit on the size of response; it also clogs up the JVM for others, putting all other concurrent requests in danger of draining the heap, even if their individual demands are modest.

The proposed technique thus does not make use of the above approach. It sends the updates using streams as response. At the client, this stream is iterated and the zip entries are fetched as jar files. It does not store any temporary zip files at the client and server's memory for every update request. The proposed approach reduces both time and memory consumption by providing a reliable and efficient use of java streams.

C. Maintenance of Logs

As a proper logging mechanism is incorporated in the proposed technique, all the steps can be tracked for monitoring proper functioning of the system. Even when the client reboots, if any discrepancy is found, it can be backtracked to a stable state without hampering its functioning by referencing the logs.

ISSN(Online): 2319-8753
ISSN (Print): 2347-6710

International Journal of Innovative Research in Science, Engineering and Technology

(An ISO 3297: 2007 Certified Organization)

Website: www.ijirset.com

Vol. 6, Issue 6, June 2017

D. Improved Security

The client does security check twice. Once when it completes downloading the package it verifies the count of the downloaded files with the server. Thus, ensuring the completeness of the downloaded package. Secondly, it again double checks the authenticity of the package by calculating its hash value and cross checking it with the server. This allows clients to be assured that the package has not gone under any interception or alteration over the network. Thus, combating man-in-middle attack by maintaining the crucial business logic intact over the network. The client is assured of the originality of the package and avoiding the package to be affected by any malwares. Thus, this approach presents a robust technique for auto updating the clients.

VII. CONCLUSION

The proposed system very efficiently proves its mettle in automating the update process at the client side. It overcomes all the inefficiencies and gambling associated with the traditional manual approach. The use of ZipInputStream makes efficient consumption of memory. The technique of hashing the jar files also allows the client to be assured about the integrity of the downloaded jars along with providing security against several serious attacks. Thus, incorporation of the proposed technique provides online update of the clients in a more universal and intelligent manner by enabling the updates to be easier, speedy and secured.

REFERENCES

[1] Q. Wang, Q. Du and G. Lin, "IPTV STB Software Update Scheme Based on SNMP," 2006 IEEE International Conference on Electro/Information Technology, East Lansing, MI, 2006, pp. 197-200. doi: 10.1109/EIT.2006.252140
[2] B. M. Luettmann and A. C. Bender, "Man-in-the-middle attacks on auto-updating software," in Bell Labs Technical Journal, vol. 12, no. 3, pp. 131-138, Fall 2007. doi: 10.1002/bltj.20255
[3] R. Clayton, S. J. Murdoch, and R. N. M. Watson, "Ignoring the Great Firewall of China," Proc. 6th Internat. Privacy Enhancing Technol. Workshop (PET '06) (Cambridge, Eng., 2006), pp. 20–35.
[4] Ethereal, http://www.ethereal.com.
[5] Internet Engineering Task Force (IETF), "PublicKey Infrastructure (X.509) (pkix)," Aug. 24, 2006, http://www.ietf.org/html.charters/pkixcharter.html
[6] Libnet, http://www.packetfactory.net/libnet/
[7] Microsoft Corporation, "Authenticode Overviews and Tutorials," http://msdn2. microsoft.com/en-us/library/ms537360.aspx.
[8] Microsoft Corporation, "Microsoft Internet Explorer 6 Resource Kit," http://www. microsoft.com/technet/prodtechnol/ie/reskit/6/part2/c06ie6rk.mspx.
[9] nabocorp softwares, "Cam2PC," http://www. nabocorp.com/cam2pc/index.php.
[10] B. Schneier, Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd ed., John Wiley, New York, 1996.
[11] D. L. Shinder, "Code Signing: Is It a Security Feature?" WindowsSecurity.com, June 9, 2005, http://www.windowsecurity.com/articles/Code-Signing.html.