Web-Based Data Migration Management Tool for Multi-Cloud Services

M. Wilhoite, J. Lieu, D. Do

Computer Science

California State University, Fullerton

College of Engineering and Computer Science

I. Introduction

The Data Migration Management Tool aims to provide users a tool for “securely” transferring files between cloud nodes. This system is meant to be used by users to manage multiple nodes and perform the following functions on files or directories: transfer, copy, and delete. The user interfaces with these functionalities via a web application.

**Midterm report - This portion of the report serves as documentation of how our system functioned as of the midterm presentation.**

II. Technology Stack

Our system uses an AWS EC2 instance to host our manager node and web page. The backend API is built and hosted on our manager node by using Python and Flask-API library. The frontend web page is hosted by the same manager node using Python’s SimpleHTTPServer. The frontend web page is built using HTML, JavaScript, and CSS.

For testing purposes, two individual employee nodes are set up on a separate EC2 server and Google Cloud VM instance. These nodes have a copy of the employee microservice. The employee microservice serves API calls for crawling(building the topography of the file system) and file transfer between nodes.

III. Communication Between Clouds

In our project we will have a lot of communication between the clouds, it could be: commands between the nodes, file transfers, or user’s information. These commands, files, or information will be a part of our HTTP APIs request body and will be processed by our microservices. We are intended to protect these HTTP APIs request payload by using SSL certificate public/private key encryption. This means that each node needs to register for the SSL certificates.

IV. System Design

Our system design is one manager node and currently two user employee nodes. The manager node hosts our frontend, backend, and SQLite database. The backend of the manager node holds the functions and database to support user registration and login.

The employee nodes are the user’s clouds that host our repository and communicates with the repository hosted on our manager node. Each employee node has a copy of the crawler API used to return directory and file metadata to the front end. The employee nodes also have a copy of the transfer API to perform the SCP to transfer files and directories to another employee cloud node.

The frontend web page interacts with our manager node for login and registration. To perform crawl, the frontend sends the target directory to the crawler on the target employee cloud. That crawler will respond with data of the files and directories under that path. To perform a copy from one employee node to another, the frontend sends the instructions with the target file and target destination of the target employee to the employee on which the target file resides. The employee with the file then performs SCP to send the file or directory on the target cloud at the target location.

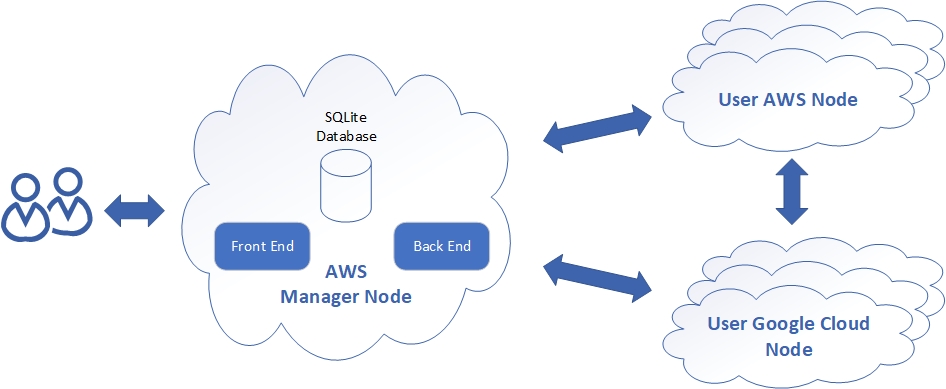


Fig. 1 Previous design of our system.

V. Front End

The frontend application is hosted on the manager node using Python’s SimpleHTTPServer. The user will see a login and registration page upon connecting to the website. The user will be able to either login to an existing account or make a new account. To register, the user will fill in the email and password fields under the registration tab. To login, the users will provide the email and password of an existing account. Successful logins will be redirected to the application page.

In the application page, users are able to crawl individual nodes to view files and directories in that node. Users are also able to copy files and directories between nodes. To crawl, users may use the expanding menu. The menu initially displays the contents of the desktop directory of the node. The user is able to expand a directory by clicking the list item. The application will display directories, signified with a plus sign on the right, and files of the crawl target.

To copy files, users will use the checkboxes to the right of the list items and the transfer buttons. Users will check one box to select the target of the transfer and another box to signify the location. After both components are selected, users may press the transfer button to send the target from one node to the location on the other node.

VI. Micro Services

We wrote two microservices that we used to process our HTTP APIs requests. These microservices are written in python using Flask API library:

*A. Users Microservice*

The user microservice is running on the EC2 instance where the front end will make HTTP APIs request call to register and user login. The register is a POST request to the endpoint ‘EC2-IP:5000/api/v1/cloud/users/register’ and it is expecting a JSON payload containing key and value for ‘Uname’ which is the username, ‘ uEmail’ which is the user email, and ‘uPassword’ which is the user password. We use SQLite database to store the user information on the EC2 node. The user password is stored encrypted using Werkzeug’s generate\_password\_hash function().

The login also a POST method with the endpoint of “EC2-IP:5000/api/v1/cloud/users/login”, this method is also expecting JSON payload that contains the key and value for “uEmail” and “uPassword”. The microservice would then use function check\_password\_hash() from the Werkzeug to validate the provided password and the hashed password in the SQLite database.

*B. Employee Microservice*

The employee microservice is running on two other separate cloud instances, one of the instances is another AWS EC2 and the other is a Google Cloud instance. The employee is in charge of doing the crawling and copying. Both of the crawl and copying features will be explained more in-depth in the utility functions section, but here is how to make requests to them:

The crawling is a POST method with the endpoint of “Employee-IP:5001/api/v1/cloud/employee/crawl” which is expecting a JSON payload that contains the key and value “paths” that tell the node where the user wants to crawl.

The copying is another POST method with the endpoint of “Employee-IP:5001/api/v1/cloud/employee/copy” which is expecting a JSON payload that contains the key and value for “sourcePath” which is the path of the file to be copied. The “destinationIP” which is IP of the node that’s receiving the copy file. The “destinationPath” is the path where the file will be placed on the destination node. The “recursive” which is a string of “True” or “False” to tell the node to copy recursively in the directory or not. The “private\_key” which should be the key string of the destination private key for copying access. However we couldn’t get copying to work when sending this key string, so we manually place the key on each node, and we set the value of “private\_key” to be the location of where we manually placed the key.

VII. Utility Functions

The utility functions of our system are the back end functions that the manager and employee nodes call to perform the core functions of the data transfer and display of the file system on the user’s nodes. The functions include a crawler, two transfer functions, and a string formatted for ssh keys.

The crawler function uses an absolute path to a specific directory to crawl the file system and return metadata of each item in that directory to be displayed on the front end website for the user. The two transfer functions currently copy a file from a source directory to a destination directory on the user’s clouds. This process uses the Secure Copy Protocol (SCP) through the Paramiko Python library. This process needs a path to the files being transferred, the destination cloud’s IP address, login username, the path to where the item will be copied, and the private key of the cloud instance. The reason there are two transfer functions is that the private key of the other cloud system is needed to use the SCP. In our current iteration, we are storing the private key of the user’s clouds in the other cloud so we can copy files from one to the other. In the next phase, we would like to eliminate the need to have to store the keys on each user node so the other transfer function relies on the private key to be passed into the function as a correctly formatted string. From the string passed in the function, a temporary duplicate of the key file is created and then the copying of the file works similar to the other transfer function using Python’s Paramiko SCP. To format the string correctly, we needed another function to form the string on the front end to send into the employee nodes for the alternative transfer function.

VIII. Problems Encountered

*A. Copy module runs as a script*

We have run into a couple of problems while developing our system that we had to work around for the system to function properly. When copying files across clouds we initially were using the function as a module and attempting to just call the function normally. However, when passing arguments into the function that were retrieved directly from the JSON data we would get a raw dump error. To work around this, we launch the function as a separate script and pass the arguments through command-line to the script, functioning similar to a subprocess for transferring files.

*B. CORS Policy Error*

One of the initial problems that we thought was a show stopper was the CORS policy restriction. When we first tried to integrate our front-end and backend together. We received an error message on our Chrome developer console saying, “Access to XMLHttpRequest at … from origin … has been blocked by CORS policy”. Upon researching about CORS, we’ve learned that CORS stood for Cross-origin resource sharing [12]. CORS is a security implementation that prevents requests to a resource to be passed between different domains. For example, if a client sends a request for a particular resource to a server on domain A. The server on domain A doesn’t have that resource so it would request another server on a different domain B that has the resource. This type of “Cross-origin” resource request is restricted by default on any system.

The default CORS policy prevents our copying functionality to work properly. Since the user will make a copy request for the resource from the front-end. Which the front-end will make APIs calls to the backend. Then the backend will make subsequence APIs call to the Employee node to handle copying of the resource.

To remove the restriction on our systems, we learned that we needed to include the HTTP header “Access-Control-Allow-Origin”, and setting the value of the header to be “\*”, which will allow our system allow the APIs resource access request from any domain. We understand that this has a major security risk, and we should only allow cross-site access to a trusted domain. But to reduce the complexity of the system for testing and development we allow cross-site resource access from any domain. For any future, production use of our system needs to specify explicitly which domain is allowed to access the resource.

Since we’re using Flask\_API library to implement our APIs calls, there is also a supporting library call flask-cors that we installed. This library allows us to configure the CORS policy with ease by decorating our Flask\_API @app.route with @cross\_origin. Allowing cross-site resource access.

*C. SSL Certificate*

At the time of implementation, we are not able to utilize HTTPS TLS/SSL encryption mechanism to encrypt and secure our API requests. Since we are using free accounts from our cloud provider, Amazon, and Google cloud. The IPs of our cloud servers are not static and change whenever we shut down or restart our cloud node. Due to the non-static IP, we are unable to register for an SSL Certificate and a Public/Private key pair.

In the production use of our system. The user needs to obtain an SSL Certificate for each node which includes the front-end, back-end, and individual employee nodes.  
 With the obtained SSL certificates, the user would be able to easily use HTTPS TLS/SSL by hosting each microservice on a production-ready uWSGI server, by running this example command from the documentation of [13], “uWSGI --shared-socket 0.0.0.0:443 --https =0,foobar.crt,foobar.key”. This command will route all the HTTP API call traffic through TLS/SSL port 443. Where foobar.crt is the SSL certificate and the foobar.key is a public key for that certificate.

IX. Plans For Final Submission

*A. General System Development*

The system as it currently works is limited in functionality. The plans we have moving forward are to develop a more robust system that removes a lot of the manual set up process that we have to do to get the system working. We plan to implement the moving and deletion of files across the user clouds. The database running in our manager node functions reasonably but it limits our plans to move forward with our planned changes and thus doesn’t provide scalability for our system. The current setup process limits the inclusion of many user nodes interacting with the manager node and would need an overhaul to accommodate this. The automation of node setup is planned to occur after the user registers for the website which will initialize the employee service on all clouds that the user connects to our system.

*B. Front End and Manager Node Changes*

The frontend application will be updated to include elements for the user to initiate the planned functionalities such as user-defined clouds and deletion. The frontend will also be moved onto an S3 Bucket using its static web hosting. We also plan to move the functionalities of our manager node on to AWS Lambda. Doing this will allow for load balancing because Lambda will spawn an instance for every user connection. It will also remove the need to have a persistent EC2 instance. The database will also be relocated onto S3 Bucket for integration with Lambda. By moving the frontend and backend onto Lambda and S3 Bucket, we remove the need to host the manager node on a persistent EC2 instance. This will improve the availability and scalability of our system.

*C. Key Management*

As part of our first demo implementation of the system, we are having trouble with designing our system on how to best facilitate the private key of an individual employee node when it is needed by other employee nodes for copying data from one node to the other.

Our initial design was to send the contents of the private key as part of the JSON data in the body of the API copy request. However, we questioned the security of our design idea and have decided to not proceed with the implementation. We have also decided to scoped out the key management aspect of our system for the first demo. We provide each node of the necessary private key by manually copying the private key to each node via SCP, and referring the key location on the system inside the JSON data body of the API copy request.

We will continue to find a better mechanism to manage our private keys, and planned to circle back to the key management aspect of our system later down the road.

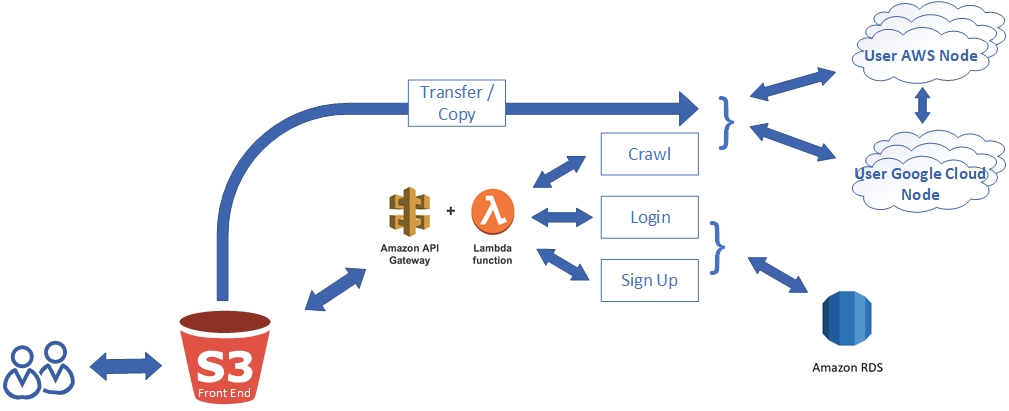
**Final report - The following portion of the report outlines how our current system functions and the changes we made after the midterm representation.**

Fig. 2 Current design of our system.

I. New System Design

Our current system design is an altered version of our previous design. We removed our manager node on the EC2 instance and split the pieces across different services. Our front end is hosted on an Amazon S3 bucket and our database is hosted on an Amazon RDS instance. We also moved our user request system from the manager node into an API Gateway with Lambda functions.

The reasons for our structure changes are because we saw how interesting and powerful AWS Lambda functions and API Gateway is from other classmates’ presentation and we want to experiment with these services as well. The second reason is that we are still experiencing CORS issues when we moved our front-end to the S3 bucket. So our solution is to migrate all of our services behind an API Gateway and use the gateway as a proxy.

*A. AWS Lambda and AWS Gateway Purpose*

The register, login, and crawl functions are hosted on AWS lambda. AWS Gateway provides an endpoint for the functions. The purpose of this change was to remove the need for an EC2 node, which needed to be running at all times. Lambda also handles load balancing, which could be accomplished by having multiple EC2 nodes. An added benefit of the redesign is that on a small scale, lambda will not charge us while using multiple EC2 nodes will.

*B. AWS S3 Bucket Purpose*

The front end host, which was previously the manager node, is now on AWS S3 Bucket. We moved the HTML, JavaScript, and CSS files onto S3 to fully remove the need for a permanent manager EC2 instance.

*C. RDS Purpose*

Since we moved our user signup and login from the EC2 manager onto the AWS Lambda function, we can not use the SQLite database to store our users’ information anymore. We decided to use Amazon RDS which is the Amazon version of the SQL database. In the future, we are also intended to implement a key management service that will query RDS for the user’s keys to the cloud instance.

II. New Functionalities

This version of the application includes new functionality for delete, transfer, and employee node information input:

*A. Delete*

The delete function requires that the user targets a file or directory and delete it from the node it exists in. To do this, the user will target an object by selecting the right-most checkbox of the row on its respective side. The user will then press the “Delete” button. The file should be removed from the node. To see the update, the user will have to refresh the directory by minimizing and reopening the directory that held the file. The delete function in our employee script is a POST method with the endpoint of “Employee-IP:5001/api/v1/cloud/employee/delete”,  
 which is expecting a JSON payload that contains the “sourcePath” which is the absolute path to the file or directory that will be deleted. The other item in the payload is “recursive” which is a string equivalent of True or False flag to determine if a file or directory is being deleted since they use different commands.

*B. Transfer*

The transfer function allows the user to move one file or directory from one node to another. It is similar to the copy from the previous version. In the copy procedure, the original copy remains in the original node, whereas in a transfer, the original is deleted. To perform a transfer, the user will select the rightmost checkbox in the target object’s row on the respective side. The user will also select the destination by checking the leftmost checkbox of the destination directory on the other side. To differentiate a copy from a transfer, the checkbox next to “Send” is used to signify whether or not the original is kept. Unchecked means the original is deleted for the transfer procedure. The user will then press “Send” to perform the action. At this point, the user should see that the target file is no longer in the original directory and is moved to the target directory. To see this update reflected on the web application, the user will have to reopen the original location directory and the new location directory. The transfer function relies on the same POST method that copy uses but we added an extra parameter to determine if a copy or transfer was to be called.

*C. How to Delete, Transfer, and Copy Function*

The back end functions rely on communications from the front end to be performed. After the system is crawled the paths of all the objects in that directory are received and stored locally on the front end. When an item is selected on the front end, and one of the actions is performed, the rest of the required parameters are calculated based on which function is being called. The copy and transfer functions rely on the Paramiko Python library to send files over the SCP protocol. These functions also rely on the corresponding key of the cloud that the file or directory is being sent to. This currently is being transferred as a string as a part of the JSON package that calls the function. With all these parameters, the corresponding function then executes as a pseudo-sub-process being executed via an OS call.

*D. Employee Node Information Input*

In the previous version, the employee nodes IPs, usernames, and keys were hardcoded into the application for testing purposes. It was only able to connect to our two predefined test employee nodes. The new update added a new page to the application after the login. On this page, the user is prompted for the IP, username, and key of two separate nodes. After this information has been inputted, the user may continue onto the application. This was done to move the application away from the testing phase and ready for release.

III. Problems Encountered

*A. CORS*

The largest difficulty we face was CORS. We had issues with sending data to our Gateway endpoint because of this and could not resolve it before the deadline. We had two solutions to bypass CORS.

Our initial solution involved using an EC2 node to act as a proxy between the browser and Gateway. To do this, we had our front end send data to a listening EC2 node, which would then relay it to Gateway to trigger lambda. This solution was successful but we decided to not go with this implementation. We wanted to completely remove the need to host our own EC2 instance.

Our other solution was to disable CORS on the browser. This worked and allowed us to move on with development, although it is not ideal. The current version of the application uses this solution. The user needs to disable CORS to use the application.

*B. Invoking Backend Functions*

When we initially were calling the functions by importing our utility script we were receiving errors that we could not determine the cause of because when running the function directly we were not receiving these errors. To remedy this issue instead of importing the functions and calling them from our employee script, we launch our utility script which has our transfer, copy, and delete functions as a pseudo-sub-process from a command-line call. We pass the parameters needed for the functions as command-line arguments and access them through sys.argv[x] arguments.

IV. Plans For Further Development

*A. User Account Functionality*

Currently, our user account feature is not utilized in the main application. The next step with this would be to store user-defined nodes in our database with the respective user. The user would only be prompted for node information on the initial access. On subsequent visits, the user will skip the node data input page and go straight to the application page, with their predefined nodes loaded. There will also be a function for the user to redefine their nodes, which will take the user back to the node information input page.

*B. CORS*

We plan to configure our front end to work with CORS while directly communicating with lambda. The use of the bypass is insecure but necessary to match the deadline. To improve the security of the application, CORS will need to be correctly configured.

*C. Employee Node Setup*

Currently, the user needs to clone the employee files from Github onto their nodes. We would like to implement a page for the user to provide their node information and have the setup be done automatically. The function will utilize a lambda function that performs SSH into their node and handles the employee set up. This process will serve to streamline our application and provide ease of use to the user.

*D. Web-based Terminal*

Another direction we could explore would be to enable the application to SSH into employee nodes and run shell commands. Doing this will allow the user to use our web application as a terminal, with the added functionalities of our predefined functions, such as copy, transfer, delete, and future additions. It would serve to provide a GUI interface for node management with tools to perform actions at a click of a button.

REFERENCES

[1] Mell, P., Grance, T, “The NIST Definition of Cloud Computing, ” National Institute of Standards and Technology. 2011.

[2] M. Almorsy, J.C. Grundy, I. Müller, “An Analysis of the Cloud Computing Security Problem”, 2010. ArXiv, abs/1609.01107.

[3] A. James, “SSH and SCP: Howto, tips & tricks”. LinuxAcademy, 2012.

[4] Z. Li, X. Li, T. Liu, S. Yang and J. Xie, "A preliminary data transfer model for private remote sensing cloud," 2016 IEEE International Conference on Big Data Analysis (ICBDA), Hangzhou, 2016, pp. 1-5. doi: 10.1109/ICBDA.2016.7509816

[5] T. Huang, K. Chu and Y. Rao, "Smart Intermediate Data Transfer for MapReduce on Cloud Computing," 2013 International Conference on Cloud Computing and Big Data, Fuzhou, 2013, pp. 9-14. doi: 10.1109/CLOUDCOM-ASIA.2013.97

[6] Y. Yu, J. Ni, W. Wu and Y. Wang, "Provable Data Possession Supporting Secure Data Transfer for Cloud Storage," 2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA), Krakow, 2015, pp. 38-42. doi: 10.1109/BWCCA.2015.44

[7] R. Tudoran, A. Costan, R. Wang, L. Bougé and G. Antoniu, "Bridging Data in the Clouds: An Environment-Aware System for Geographically Distributed Data Transfers," 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Chicago, IL, 2014, pp. 92-101. doi: 10.1109/CCGrid.2014.86

[8] C. Jayashri, P. Abitha, S. Subburaj, S. Y. Devi, Suthir S and Janakiraman S, "Big data transfers through dynamic and load balanced flow on cloud networks," 2017 Third International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB), Chennai, 2017, pp. 342-346. doi: 10.1109/AEEICB.2017.7972328

[9] V. Dubey and P. Agrawal, "Cloud computing and data management," 2016 Symposium on Colossal Data Analysis and Networking (CDAN), Indore, 2016, pp. 1-6. doi: 10.1109/CDAN.2016.7570892

[10] C. B. M. Lek, O. B. Yaik and L. S. Yue, "Cloud-to-cloud parallel data transfer via spawning intermediate nodes," TENCON 2017 - 2017 IEEE Region 10 Conference, Penang, 2017, pp. 657-661. doi: 10.1109/TENCON.2017.8227943

[11] Cloud Fare. (2019). What Happens in a TLS Handshake? | SSL Handshake. [online] Available at:<https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/>

[12] Flask-cors.readthedocs.io. (2019). Flask-CORS — Flask-Cors 3.0.7 documentation. [online] Available at: https://flask-cors.readthedocs.io/en/latest/ [Accessed 1 Nov. 2019].

[13] Flask-cors.readthedocs.io. (2019). Flask-CORS — Flask-Cors 3.0.7 documentation. [online] Available at: https://flask-cors.readthedocs.io/en/latest/ [Accessed 1 Nov. 2019].

[13] Medium. (2019). 3 Ways to Fix the CORS Error — and How Access-Control-Allow-Origin works. [online] Available at: https://medium.com/@dtkatz/3-ways-to-fix-the-cors-error-and-how-access-control-allow-origin-works-d97d55946d9.

[14] Medium. (2019). API-Gateway S3 Proxy. [online] Available at: https://medium.com/@JCDubs/api-gateway-s3-proxy-a72e398b4d03.

[15] PyLenin, f. (2019). AWS Lambda Deployment Package in Python. [online] Youtube.com. Available at: https://www.youtube.com/watch?v=rDbxCeTzw\_k [Accessed 5 Dec. 2019].

[16] Stack Overflow. (2019). Disable same origin policy in Chrome. [online] Available at: https://stackoverflow.com/questions/3102819/disable-same-origin-policy-in-chrome.

[17] www.boxuk.com. (2019). Enabling cross-domain access in CloudFront | Insight | Box UK. [online] Available at: https://www.boxuk.com/insight/enabling-cross-domain-access-in-cloudfront/.

[18] Pymysql.readthedocs.io. (2019). Examples — PyMySQL 0.7.2 documentation. [online] Available at: https://pymysql.readthedocs.io/en/latest/user/examples.html.

[19] Docs.aws.amazon.com. (2019). Tutorial: Configuring a Lambda Function to Access Amazon RDS in an Amazon VPC - AWS Lambda. [online] Available at: https://docs.aws.amazon.com/lambda/latest/dg/services-rds-tutorial.html.