College of Computing, Georgia Institute of Technology

CS6238 Secure Computer Systems

Secure Distributed Data Repository (SDDR)

Project 2

David Musielewicz

6083357254

david.musielewicz [at] gatech.edu

M.S. in Information Secuirty - DL

Chidong Xue

902885919

cxue8 [at] gatech.edu

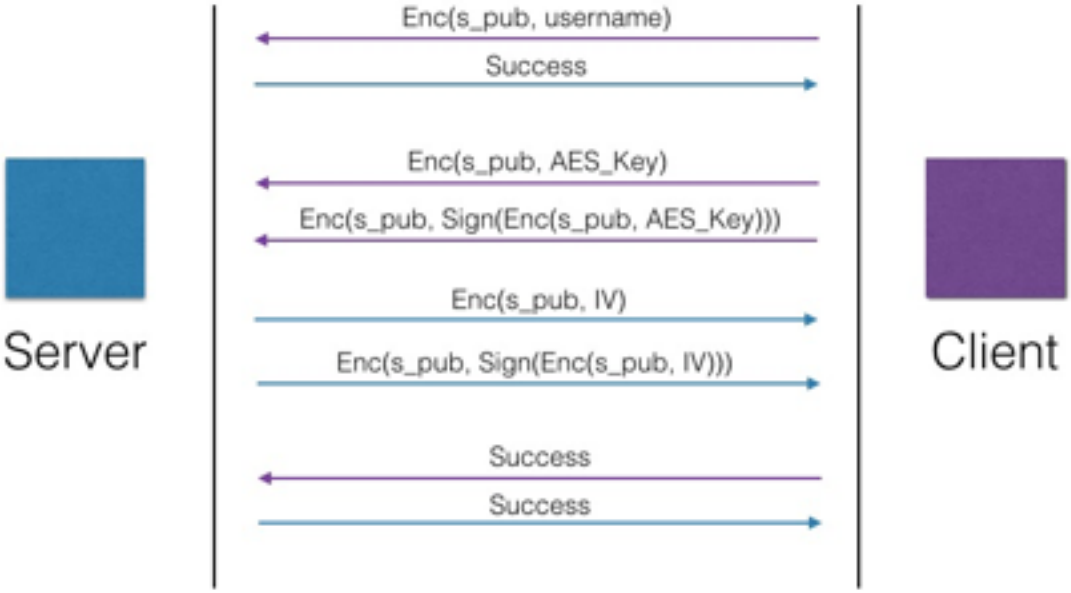
M.S. in Information Secuirty

2014.12.02

1. Protocol Details

To implement this secure file transfer service, each node (servers and clients) all require a keystore file created using java's keytool. All keystores must contain and trust the CA certificate along with a certificate of all other trusted nodes signed by the CA.

Start-session: A client initiates a session with a server by first sending a username encrypted with the server's public key. The server verifies the username as an alias in the keystore and acknowledges receipt. The client then generates a random AES session key, encrypts it with the server's public key, signs the encrypted data, encrypts the new signature, and sends both the encrypted key and encrypted signature to the server. The server validates the signature to verify the client is indeed who they claim to be. If valid, the server generates a random IV and encrypts it using the client's public key. The server then generates a signature of the encrypted IV, encrypts the signature, and sends both the encrypted IV and signature back to the client. The client then verifies the server signature. If all signatures are valid, the connection is established and future communication is encrypted using symmetric encryption with the established session key. Below is a depiction of this interaction:



Delegate: When a client specifies that a delegation token is to be created, the request and necessary details are sent to the server. The server checks if the client owns the file being delegated by parsing the Metadata object list. If the user owns the file, a delegation token is created. This is based on the fact that the client was validated during the initial session creation,

and therefore this is a valid request. Below is the format used for the tokens stored locally by the server:

<UID>:<Client>:<Expire Date>:<Propagation>:<Access>

If the client is not the current file owner, the server parses all delegation tokens for a valid propagation token. If the server locates a token with propagation rights for this user and UID, a new token is generated; otherwise, the client receives an access denied.

Get: A client first sends a get request to the server specifying the filename or UID. The server locates files owned by other users by UID and files owned by the client by filename. If the user does not own the file, the server parses the list containing all delegation tokens looking for a token allowing the client to view the file. The token must be for the UID and requesting client, have 'get' or 'both' access, and still be within the valid time window. If the user is the owner of the file or there is a valid delegation tokens, the server transfers the file to the client. The request is denied otherwise.

Put: A client is able to put a file on the server using the security flags none, integrity, and confidential. A client sends the server the filename and desired security mode. The server generates a unique identifier (UID) by using SHA1 to hash the client's username appended with the filename. This ensures that each file is unique for each user. If a client wishes to overwrite another client's file, they must specify the UID instead of the filename. The server checks for a proper delegation token in its delegation list. The token must be for the UID and client, have 'put' or 'both' access, and not be expired. The server then adds or updates the file's information as a metadata object to an internal list. The metadata consists of the following structure:

<UID>:<Filename>:<Owner>:<Security Mode>:<Security Parameter>

The security parameter is a null value if the security mode is 'none', the file's signature if the mode is 'integrity', or an AES key encrypted with the server's private key if 'confidential'. Finally, the file is stored to the local disk as the UID and (potentially) encrypted with the generated AES key.

End-session: The client stores a SHA1 hash of each file together with the file's security mode each time the it receives a file from the server. The application then checks all received files' hash values on session termination to see if any have been modified. Copies of all altered files are then sent to the server before ending the session. The server updates these files in the same secure mode as before.

2. Evaluation

We ran PMD and FindBugs plugins in Eclipse against our application to ensure there were no security flaws in the source or byte code. FindBugs had no findings while PMD reported a significant number of violations. After parsing these violations we corrected ones related to design and security flaws that may cause potential issues, and ignored violations related to syntax and coding style.

To evaluate the overhead of security we performed multiple timing tests of all functions as well as evaluating the required overhead in source code and JAR files. Upon completion of the secure application, we duplicated our program and removed all aspects of security. This included removal of all communication encryption, integrity and confidentiality checks, and delegation functionality. This resulting application only provided basic get/put functionality and allowed users to perform functions on any requested file. All timing tests were conducted by running the server and client on the localhost to reduce timing variance in routing devices. Each application was given a one gigabyte memory pool and all files were removed from the server prior to testing.

In the end the basic application (both server and client code) was 516 lines of code, versus 1,247 lines for the original project. The compiled client and server JAR files were reduced from 46K collectively to 11K; nearly a 77% size reduction. Although a difference of 34K may seem insignificant, most applications are significantly bigger than this project. If we assume the relationship between security and application size is a linear relationship, we see that designing security into the application produces a significant amount of overhead in file size and resources required.

In addition to performing a static comparison of the applications, we performed multiple timing tests during runtime as well. We performed two sets of timing tests; one testing the security overhead added by function implementation, and one specifically evaluating the overhead in file transfer. Below are our results:

	Basic Application	Secure Application
Session Initialization	19 ms	150 ms

Secure Application:

	1KB	1MB	5MB	50MB
Put - None	2 ms	37 ms	141 ms	1606 ms
Put - Confidential	5 ms	47 ms	205 ms	2151 ms
Put - Integrity	10 ms	56 ms	188 ms	1726 ms
Get - None	6 ms	35 ms	152 ms	1450 ms
Get - Confidential	8 ms	43 ms	198 ms	1939 ms
Get - Integrity	15 ms	63 ms	215 ms	2039 ms

As expected, we see that storing a file without any secure parameters requires the least amount of time to complete. Interestingly, using the put command with the integrity flag takes the longest time to complete for small files, but is overtaken by confidentiality as the file size grows beyond 1 megabyte. This are not the same results when executing a get command for a file. We see that instead, integrity takes longer than confidentiality as files grow larger. This could be a result of our decryption function taking less time than our encryption. Overall, adding security features to these files adds an overhead that diminishes as file size gets larger. We see the largest discrepancy between timings for 1 KB files; one instance taking five times as long. As file size grows to 50 MB we see this gap reduce to only a 25% overhead. There is a slight overhead in the 'get' function because the server must first perform an access check to determine if the user has access to the file.

Basic Application:

	1KB	1MB	5MB	50MB
Put	1 ms	9 ms	65 ms	321 ms
Get	1 ms	17 ms	86 ms	443 ms

Comparing the results of our basic application to our secure one, we see that adding security throughout the project results in five times the execution time. This is a result of the access checks and network encryption overhead added by security.

After performing timing comparisons between our secure and basic application it is clear that adding security throughout an application results in significant overhead overall. This must be a strong consideration when developers are designing applications for consumer use. While security is important, its implementation should be optimized to prevent and reduce unnecessary overhead as much as possible.

3. Limitations

We were able to successfully implement all required features. However, we decided to implement the server and client as single-threaded processes. This results in only one client being able to connect to the server at any given time. This could lead to potential scaling issues if this application were implemented in a corporate environment and expected to receive concurrent traffic. Additionally, this design forces the server to act as a broker between clients when delegating tokens. However, this model also allows for strict network controls that restrict clients from needing to communicate with each other across network boundaries.

4. Individual Contributions

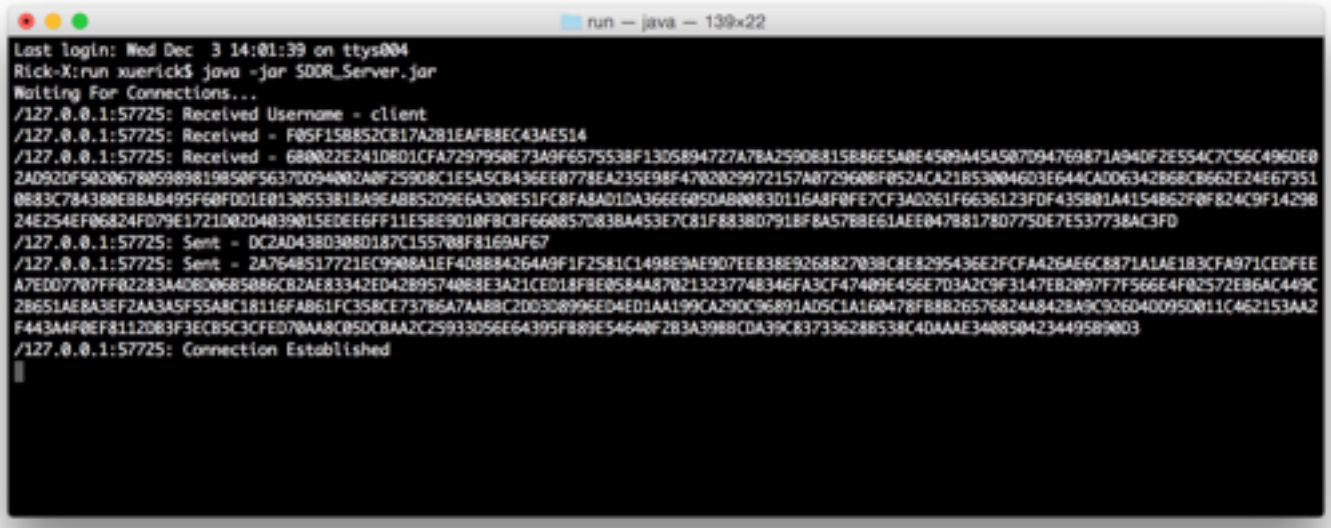
Both team members were part of the initial design for all features. David was responsible for the initial framework and coding structure for all functions. This included user input sanitation, and proper logic flow. David also implemented all communication between nodes. Chidong programmed and ensured that all files were properly encrypted/decrypted and updated upon session termination, and he performed all security checks and performance evaluations of the application.

Appendix: Examples

1. Login and Start-session

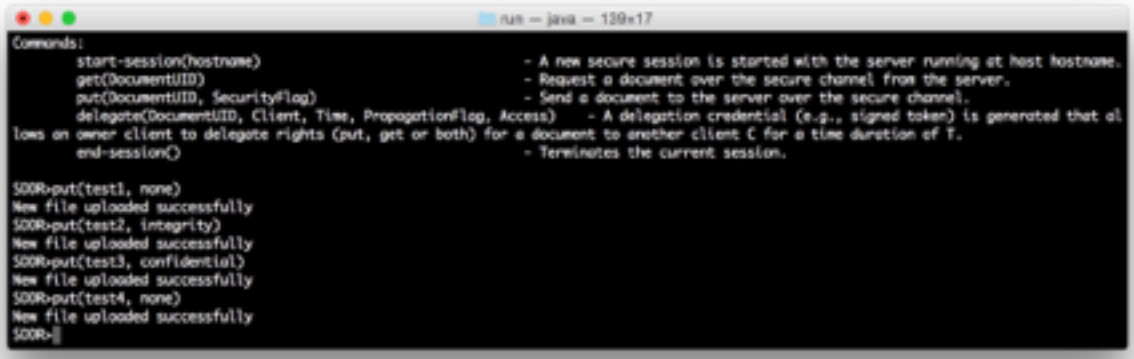
Client side:

Server side:

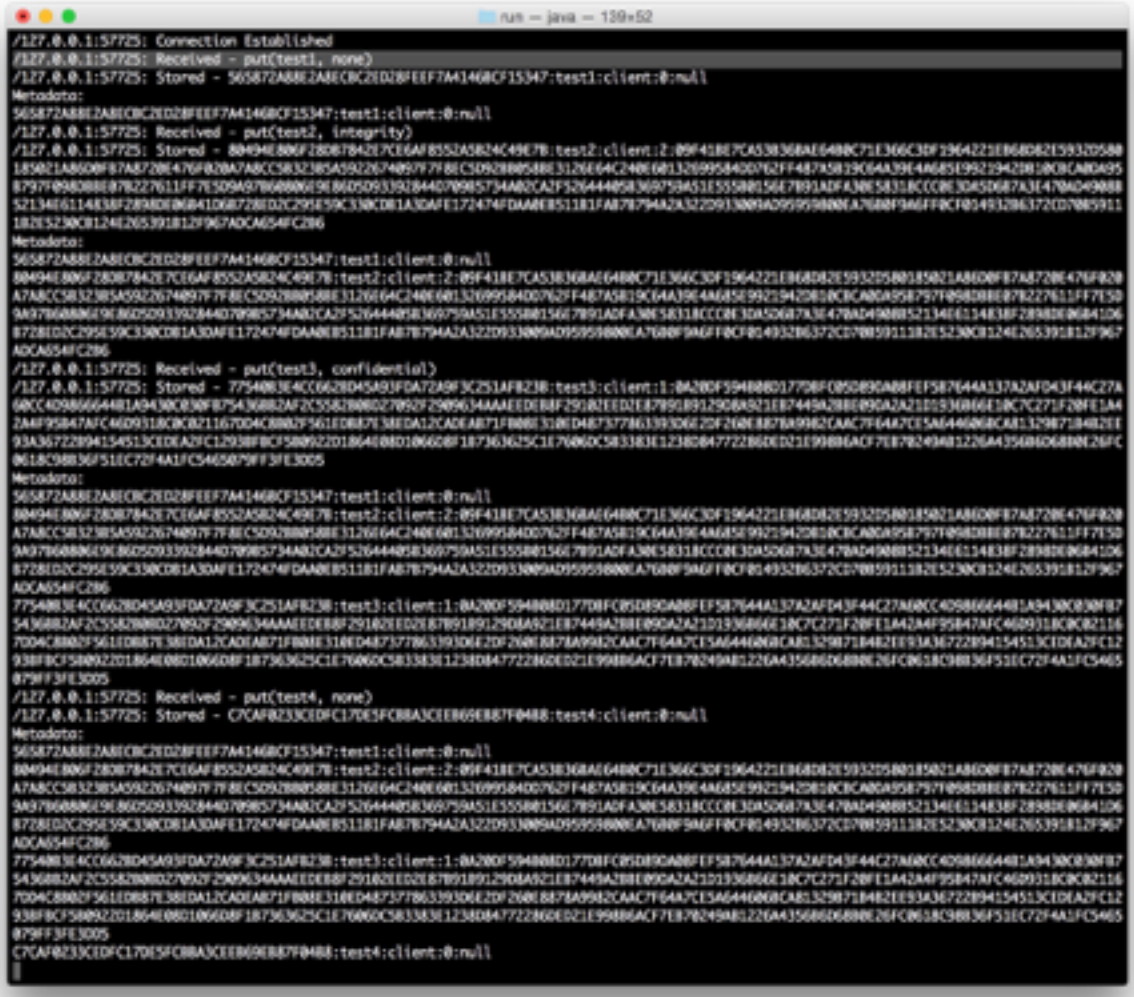


2. Put Document

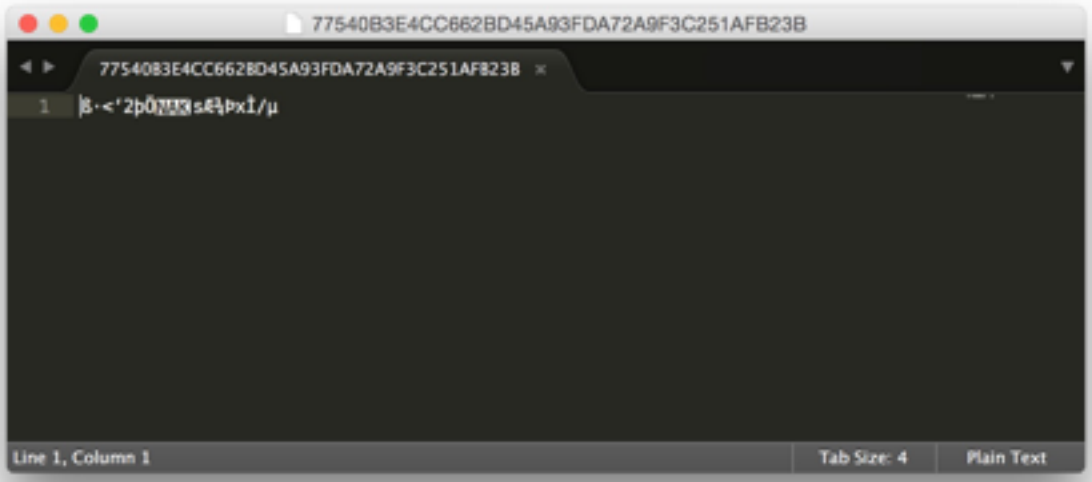
Client side:



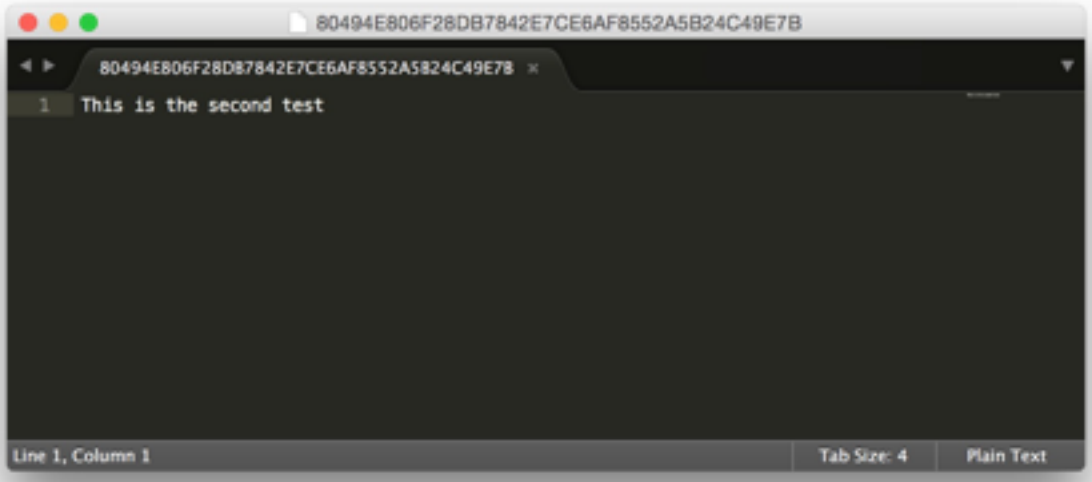
Server side:



—> File test3 is stored as 77540B3E4CC662BD45A93FDA72A9F3C251AFB23B in server, with confidential mode:

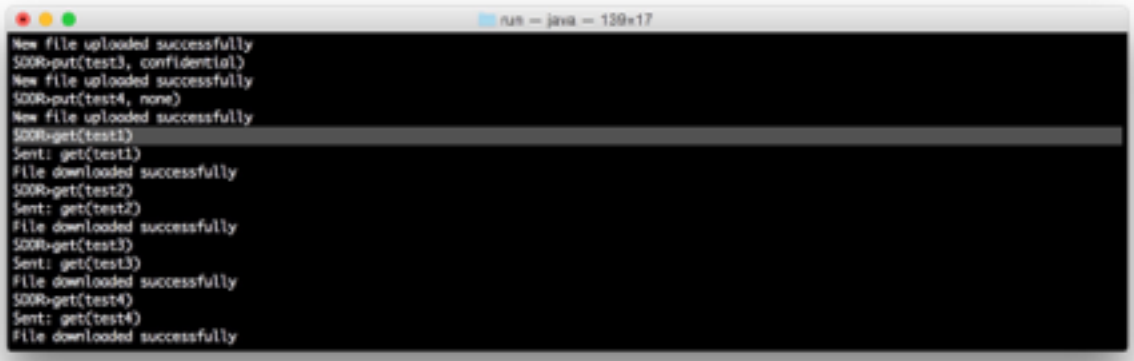


—> File test2 is stored as 80494E806F28DB7842E7CE6AF8552A5B24C49E7B in server, with integrity mode: (integrity check on this file is demonstrated later in Get Document)

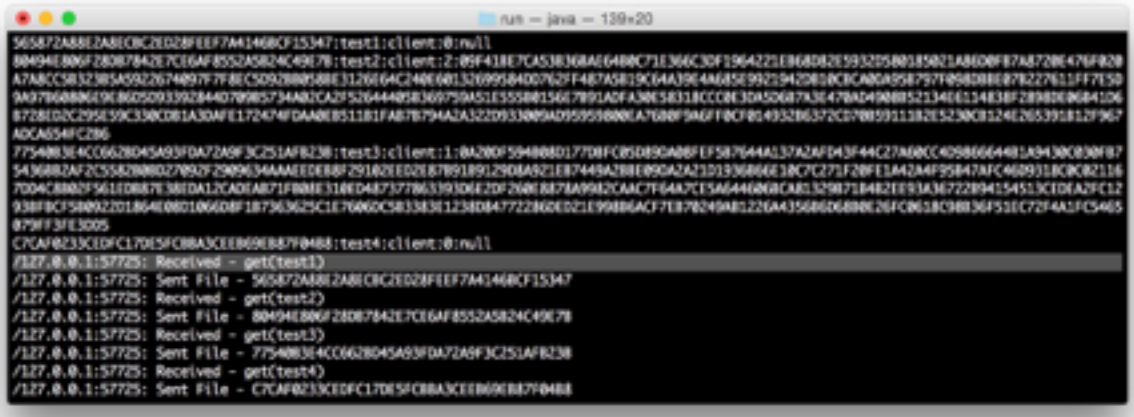


3. Get Document

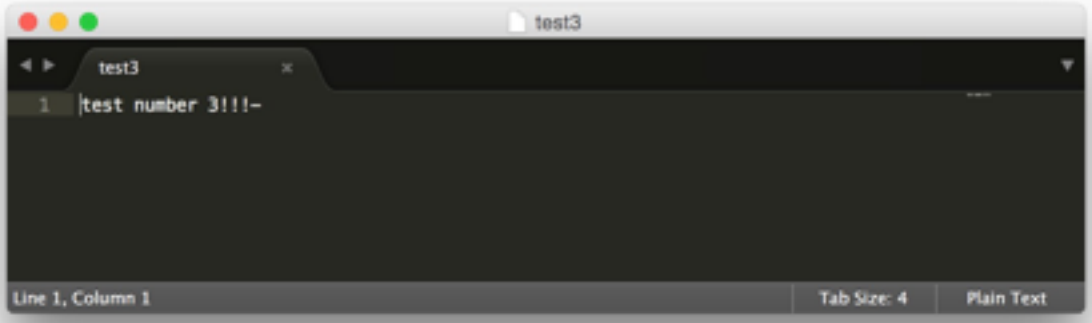
Client side:



Server side:

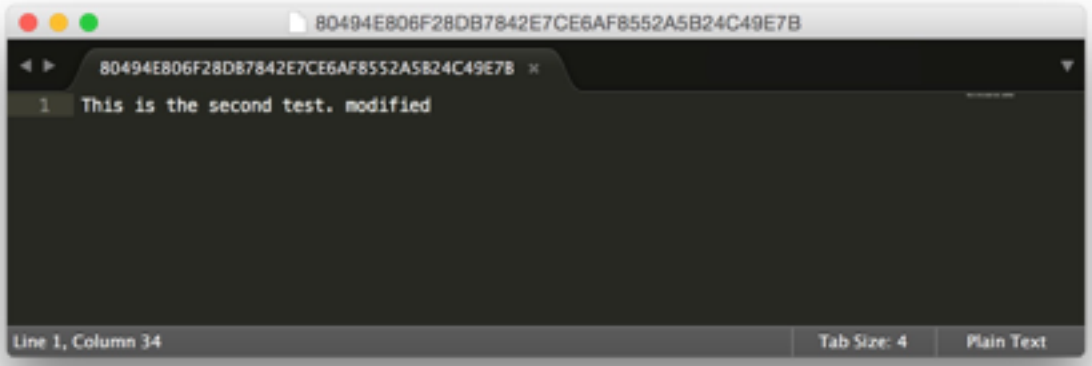


—> The confidential file test3 is decrypted and saved locally:

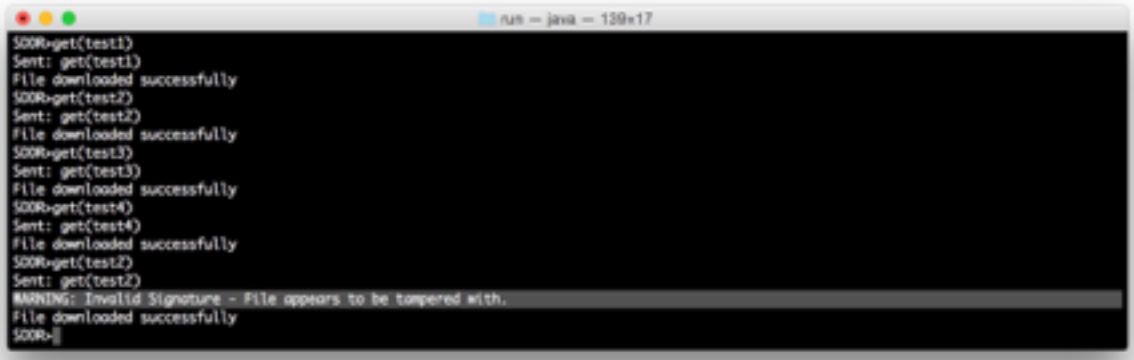


—> For the integrity file test2, we first modify it on the server:

(modify 80494E806F28DB7842E7CE6AF8552A5B24C49E7B on the server)



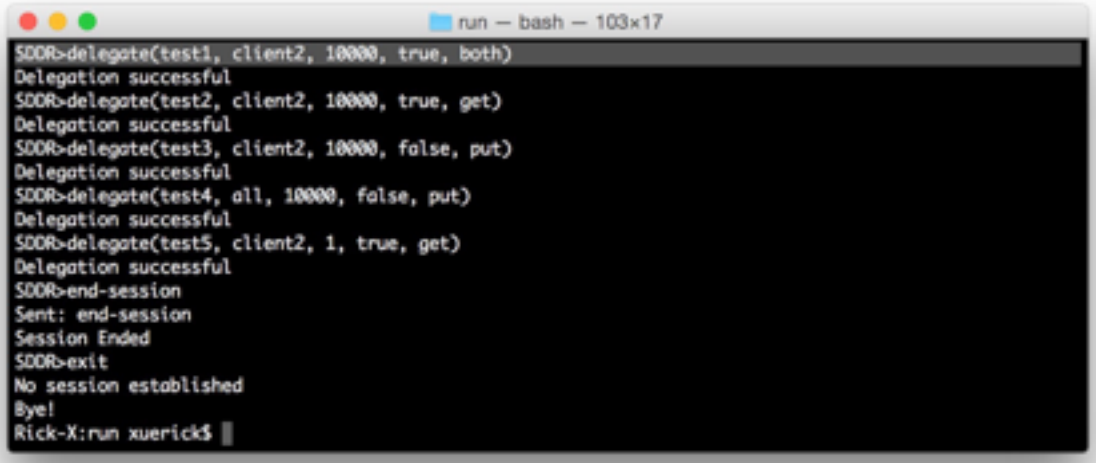
Then, try to retrieve the file again. System gives a warning to the client that it has been modified:



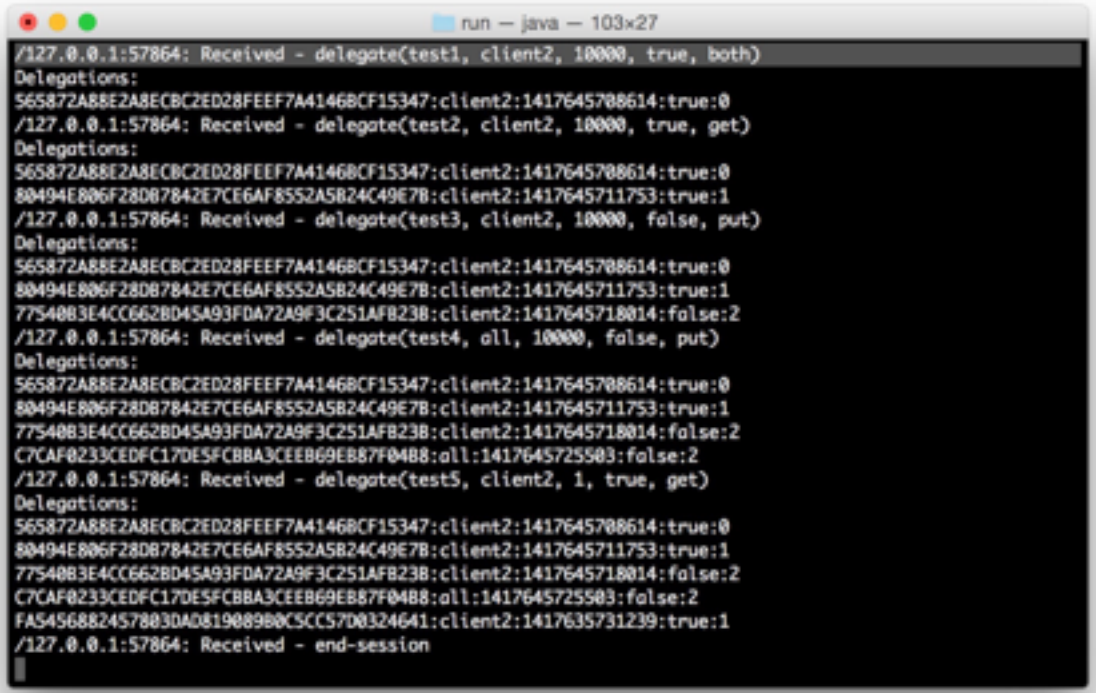
4. Delegation

4.1 Grant delegation

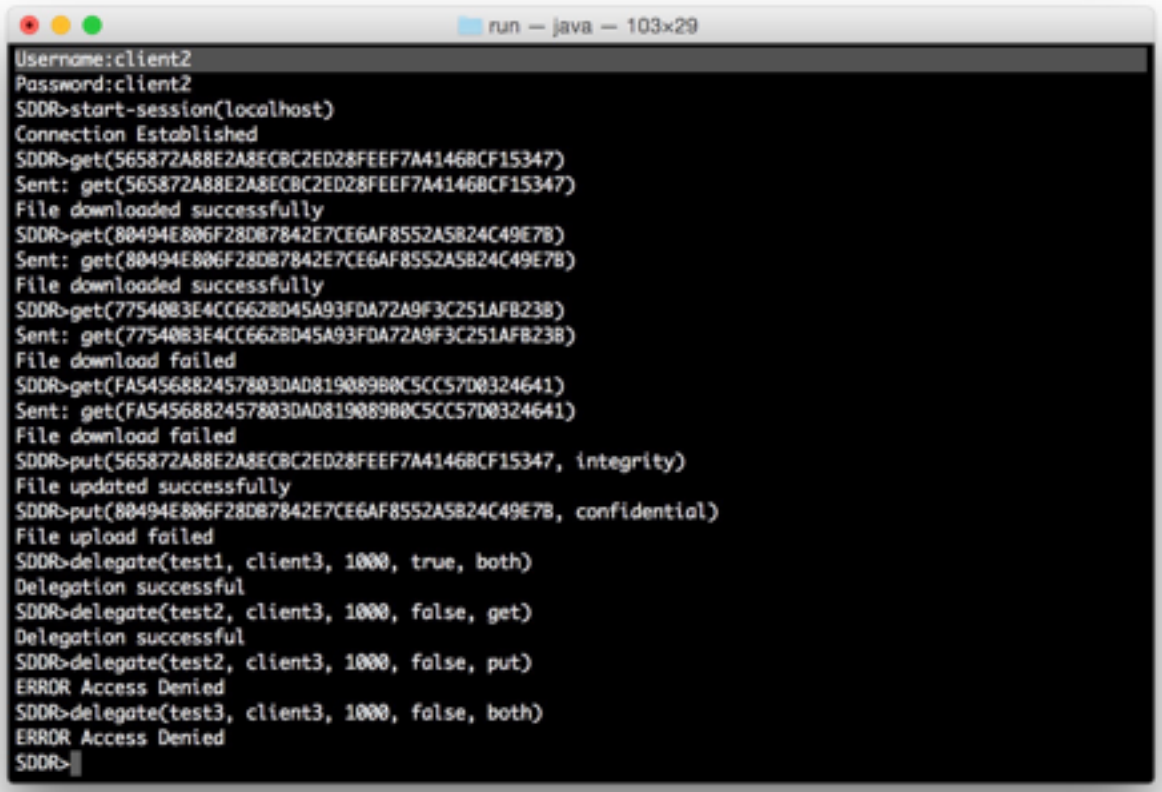
Client side:

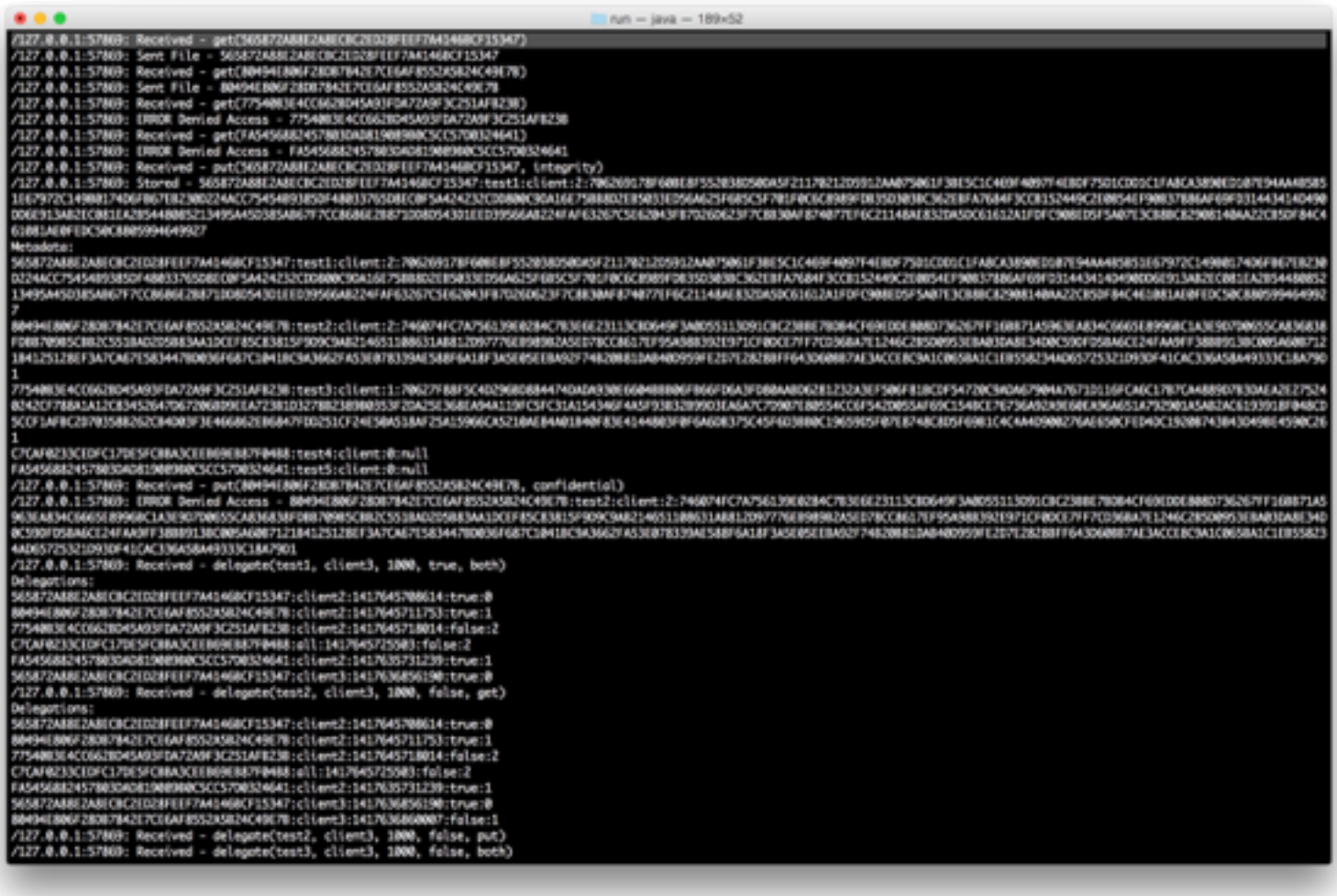


Server side:



4.2 Performing delegation





—> Explanations:

After ‘client’ grants delegations, login as client2.

(1) File 565872A88E2A8ECBC2ED28FEEF7A4146BCF15347 (test1), and file 80494E806F28DB7842E7CE6AF8552A5B24C49E7B (test2) can be downloaded successfully, since client2 has ‘get’ access for these files from the file owner.

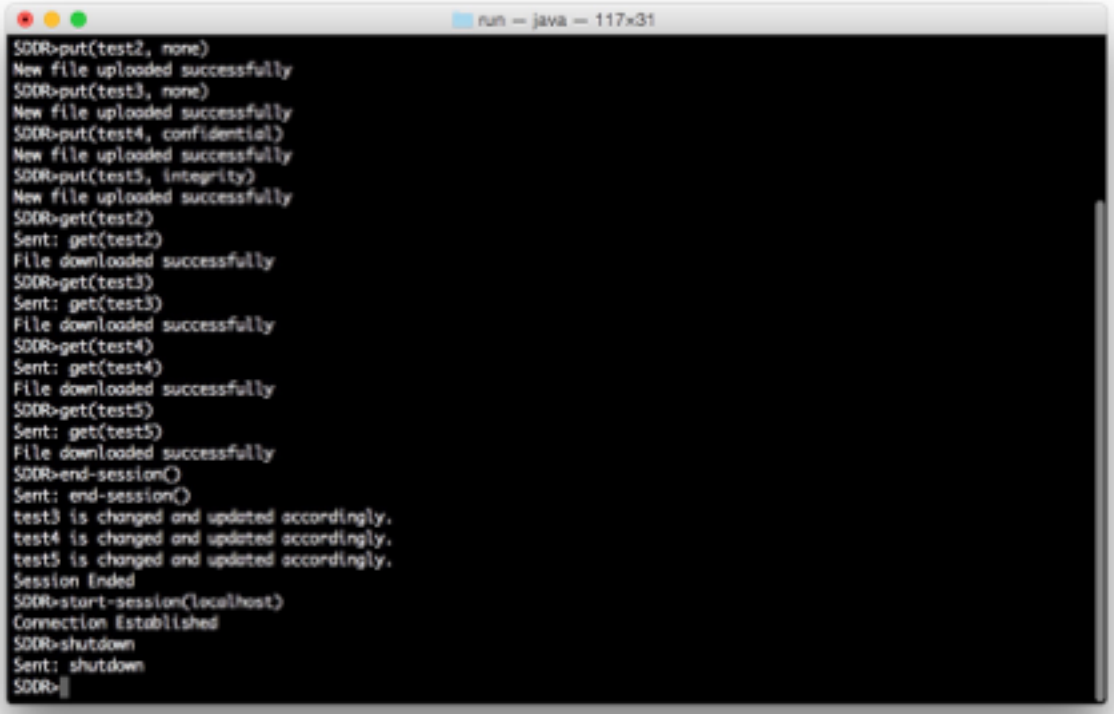
(2) File 77540B3E4CC662BD45A93FDA72A9F3C251AFB23B (test3) and file FA5456882457803DAD819089B0C5CC57D0324641 (test4) cannot be downloaded, since client2 only has ‘put’ delegation for these files.

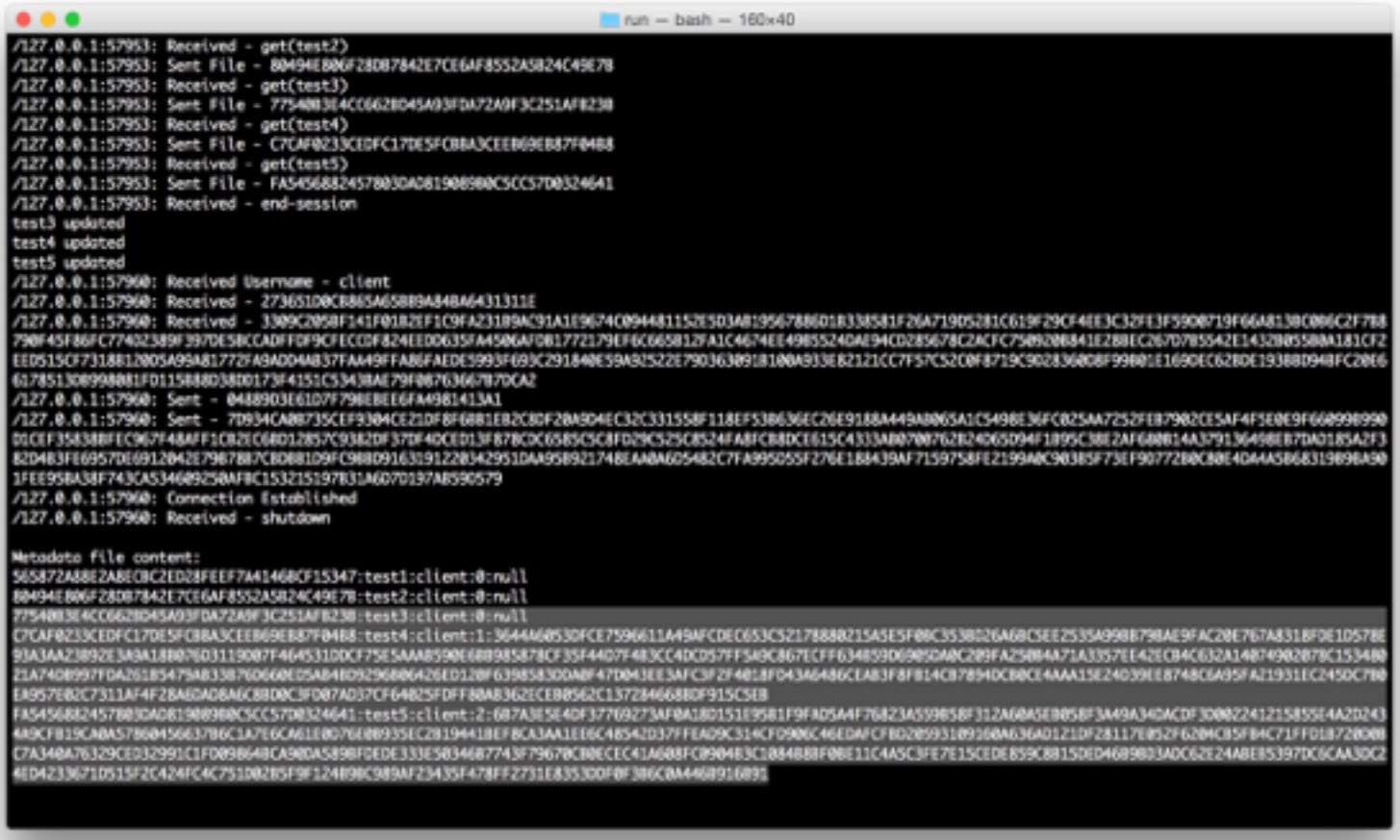
(3) File 565872A88E2A8ECBC2ED28FEEF7A4146BCF15347 (test1) can be put by client2 while 80494E806F28DB7842E7CE6AF8552A5B24C49E7B (test2) cannot.

(4) File test1 and test2’s delegation rights can be propagated with the same delegation right from the user ‘client’. File test3’s delegation right cannot be propagated.

(5) However, file test2 cannot be propagated with the delegation right ‘put’, since client2 only has the ‘get’ delegation right from user ‘client’

5. End-session and Updating





—> Upload test2, test3, test4, and test5, with security mode none, none, confidential and integrity, respectively. Then download these 4 files, and modify test3, test4, and test5.

After executing end-session(), system updates test3, test4, and test5 to the server, with the original security mode.