ECE297: MILESTONE 4

Team: Orange

Members: Jerry Wang (1000009457), Jung Yeon Kwon (1000924556),

Chee Loong Soon (999295793)

Submission Date: 2014.03.30

Executive Summary

Adobe, a software company was hacked recently, resulting in a leak of 150 million of their users' personal information, accounts, and passwords. The hackers were able to breach Adobe's servers by exploiting the simplicity of Adobe's security algorithm, whereby it uses the same key for both encryption and decryption. Our team, Team Orange is offering a more secure software database as a solution to prevent future hackers from exploiting the current Adobe's simple security algorithm.

Our database implements the public key encryption security algorithm, which is one of the most prevalent and reliable security algorithms today, specifically the Solitaire Encryption Algorithm by Bruce Schneier, which is a variant of this public key encryption. The main advantage of using the public key encryption is that the key for decrypting data lies solely in the hands of Adobe; there is less risk for a third party to obtain the decryption key. Success of the public key encryption can be seen through how widely it is still currently used as a security algorithm.

Given the size of Adobe's user-base and, correspondingly, the large amount of data Adobe has to deal with, speed remains a key component of our database implementation. To test the speed of our database, we used SQL databases as a baseline to compare our database to. We tested our database for its performance in completing routine user commands and the results, which are included in the performance evaluation, showed that our database implementation is at least as fast as SQL servers in dealing with the volume of data that Adobe is accustomed to.

Robustness is an important factor in determining whether or not an implementation of the database is secure. We ensure our database is robust by using only widely use algorithms that have lasted for many years and have proven to be robust. To be specific, we used the Solitaire Encryption Algorithm by the famous Bruce Schenier which is a form of the public key encryption. In addition, we also used LEX & YACC for parsing the command line arguments. These robust algorithms in our database are highly secure and easier to maintain due to its popularity. As a result, this also reduces its cost in implementation.

The prevalence of electronic technology today means the privacy and security of modern consumer is becoming a big issue. With the implementation described above, Adobe can ensure the safety of their client's information and maintain an efficient and cost-effective database.

1 Table of Contents

2 Introduction

Adobe is an American computer software company that focuses on the production of multimedia and creativity products [1]. Recently, Adobe's user database was hacked and a large portion of user data stolen. Among the information stolen includes customer's names, payment card numbers and other various data pertaining to customer orders [2]. As of November 7th, 2013, a purported number of up to 150 million users were affected by the hack [2]. This large scale hack shows a serious flaw in the user database of Adobe and there exists plenty of room for improvement [3].

Adobe's database is currently based on Structured Query Language (SQL) [4]. Since SQL is a general-purpose database programming language, it contains more functionality than Adobe's user database needs and introduces unnecessary complexity. This creates the problem that bugs may arise because of the added complexities. By exploiting bugs in the system, hackers can compromise the system which would compromise user data. A robust database will contain only the features Adobe requires.

An additional problem with Adobe's database is their employed method of encryption for their data. They used a form of encryption called symmetric key encryption, whereby the same key is used to both encrypt and decrypt data. This means that the same key has to be shared to the public in order to decrypt encrypted information. This allows hackers to compromise databases with large amounts of user data [5]. This is a great flaw in the database and should be addressed by the implementation of a stronger encryption system.

We have implemented a secure database that would resolve these problems. Our secure database will store private and confidential information such as customer names and payment card numbers as these information are highly susceptible to hack [2].

To address the issue of SQL, our database will tailor to the needs of Adobe and the functionality they require. This reduces the amount of redundant functionalities and keeps the code simple and robust. Less bugs means less vulnerabilities that can be exploited [6].

The problem of using symmetric key encryption can be addressed simply by replacing the current method with a public-key encryption. The public-key encryption method adds a layer of security that makes it harder for hackers to break leading to a more secure database overall. We have implemented this public-key encryption using the popular Solitaire Encryption Algorithm by Bruce Schneier [29].

3 Software Architecture

The software architecture of our database is demonstrated using three different Unified Modeling Language (UML) diagrams. The Use Case Diagram shows how an Adobe Client and Adobe Manager may use our database. The Component Diagram shows the major components of our database. The Sequence Diagram shows the sequence of instructions that occur in order to perform an operation with our database. You may refer to Appendix B for specifics regarding UML diagrams.

Figure 1 shows the use case diagram of our Summary Level secure database system for Adobe [7]. An Adobe Client can use his or her account to order products and view their product history whereas an Adobe Manager can broadcast information to its users and query information about its users. Confidential information is encrypted in the database. Refer to Appendix A for the written use case for this diagram.

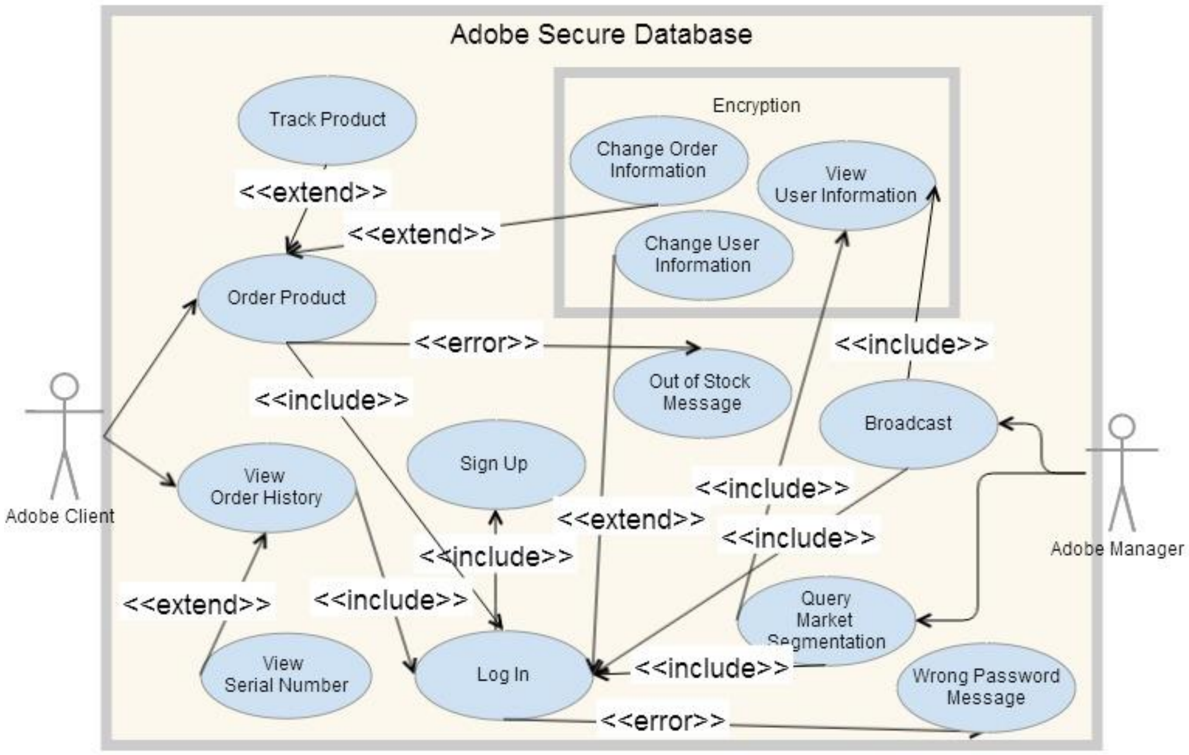


Figure 1: UML Use Case Diagram

Figure 2 shows the major components of the secure database software. The Client subsystem on the left of Figure 1 deals with providing a user interface to the client. It encrypts its information using the Public Key Encryption algorithm before communicating the encrypted information with the Server subsystem using the Transmission Control Protocol (TCP), which is supported by its Client Library. The Server subsystem is used to update data storage with its database. This subsystem is completely invisible to the user and is set up by the configuration file. Both subsystems contain utilities to support similar functionalities. The log file is used for storing operation messages, which will be useful when an error occurs.

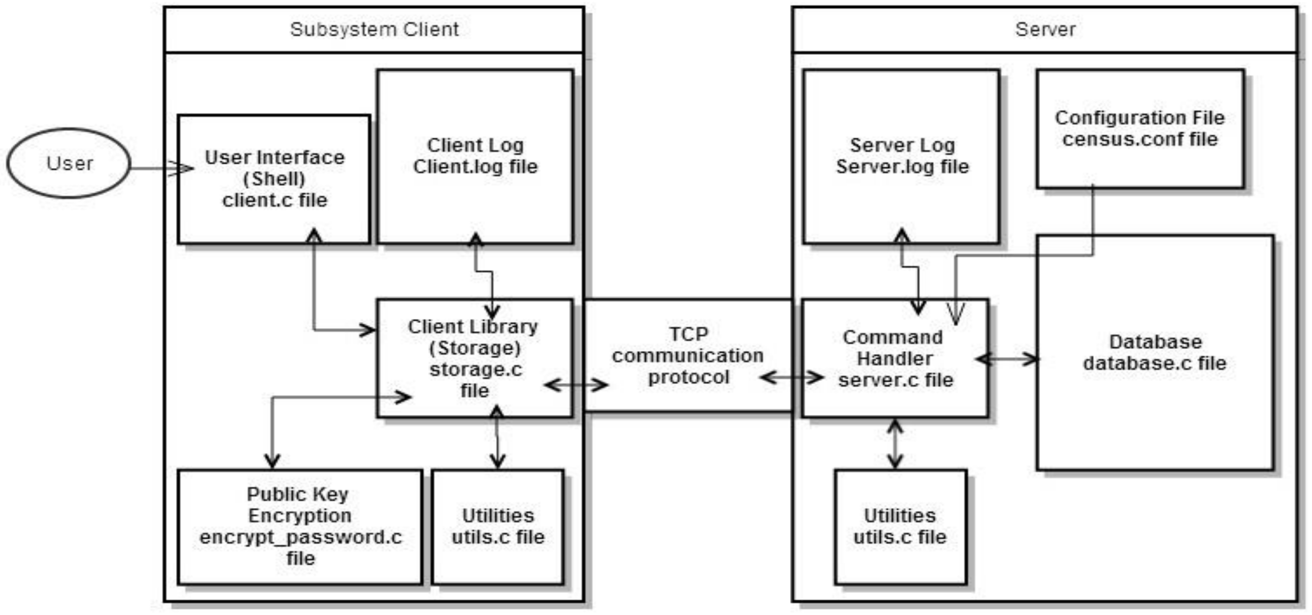

Subsystem Client
User
User Interface (Shell) client.c file
Client Log Client.log file
Client Library (Storage) storage.c file
Public Key Encryption encrypt_password.c file
Utilities utils.c file
TCP communication protocol
Server
Server Log Server.log file
Configuration File census.conf file
Command Handler server.c file
Database database.c file
Utilities utils.c file

Figure 2: UML Component Diagram

Figure 3 demonstrates the order of the interaction between the client and the server of the secure database software. It starts with the server setting up the database based on configurations settings in the configuration file. In this example shown in Figure 3, the client and server communicate via the client library interface until an error occurs, whereby the client is disconnected and removed from the server’s list of connected clients. The client will need to reconnect to proceed with further operations, until it decides to disconnect safely and simply exit without notifying the server. The server program only ends once it receives an interrupt signal, which marks the end of the server’s operation.

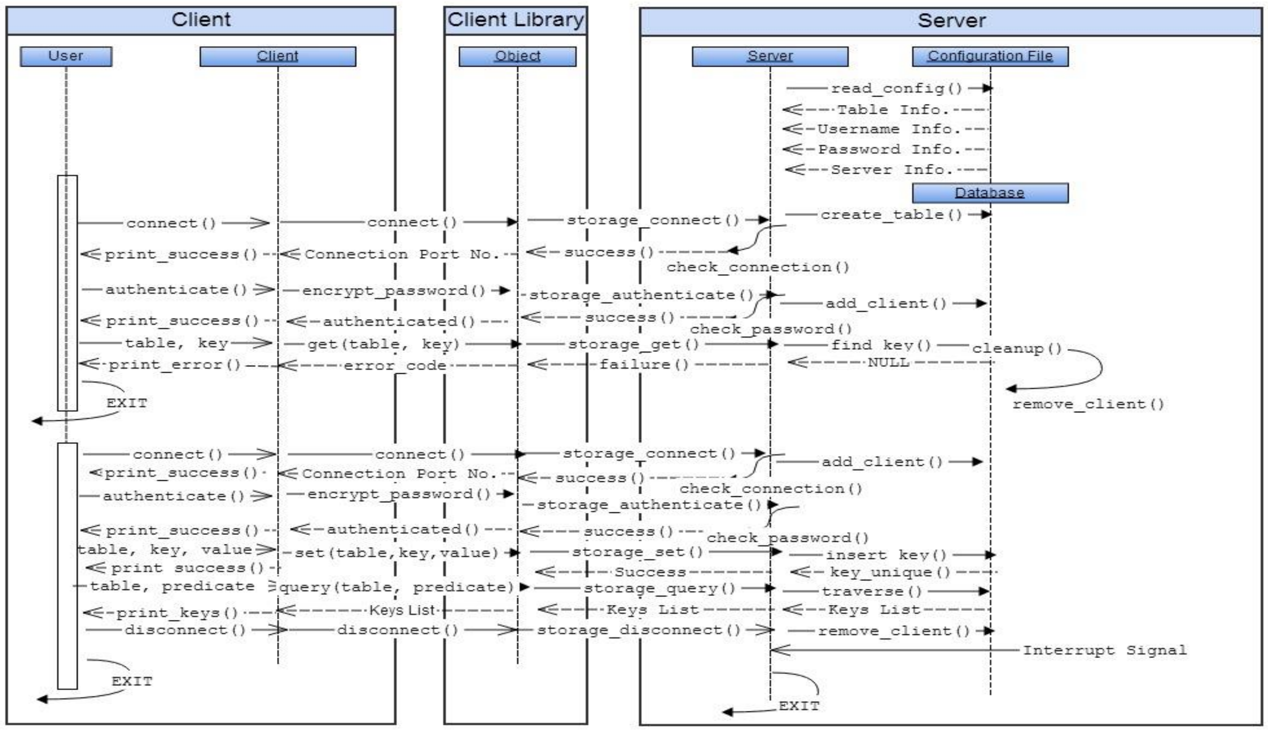


Figure 3: UML Sequence Diagram

4 System Requirements

The secure database must implement all functions without breaking its constraints. It also tries to reach its objective based on a metric that we introduce for each different objective.

4.1 Functions

- The server shall implement a shell for multiple users to interactive with concurrently [8].
- The server shall check the configuration file for proper parameters [9].
- The server shall be able to connect multiple users concurrently [8].
- The server shall be able to authenticate multiple users concurrently [9].
- The server shall be able to support tables with multiple configurations [10]
- The server shall be able to get existing records linked to a specific key and table [8].
- The server shall be able to insert new records into the database using a corresponding key and table [8].
- The server shall be able to delete existing records concurrently [9].
- The server shall be able to query existing records for information concurrently. [10]
- The server shall be able to disconnect users concurrently [8].

- The server shall be able to log data processed and functions activated in proper order [8].

4.2 Objectives:

We have to account for both user objectives and programmer objectives.

4.2.1 User Objectives:

- The database should be easy to use, measured by the number of characters a user needs to type in order to perform an operation. The lower the number of characters, the easier the database is. It should only require a maximum number of 100 characters [6].
- The server should have a lower cost of implementation than SQL, measured in Canadian dollars. SQL has a licensing cost of about $6,874 per processor, which will be our maximum cost [11].
- The server should be efficient to use by users. Our implementation should not take the user less time to use a command. The less the time taken for an operation to complete in seconds, the more efficient it is. It should take less than 4 seconds, the operation time of SQL [12].

4.2.2 Programmer Objectives:

- The server should support large volumes of user data. The server's time taken to perform a search or insert should remain constant regardless of the size of data [13].
- The server's time taken to perform a query operation should only increase linearly with the size of data.
- The server should be able to support a large amount of users connecting to the server concurrently. With Adobe having millions of subscriptions, it is reasonable to say a few thousand users are using Adobe products at any one time. Our server must be able to support at least 1000 users concurrently. The more users our server can support without exceeding the 4 seconds user operation, the better it is.
- The server should be as simple as possible, including only functionalities that Adobe needs. The lower the number the functionalities, the better [6].
- The server should catch and output error messages if the user enters an invalid information. The more types of error messages that is caught and outputted, the better the server. These error messages are grouped into 7 different categories listed in the storage.h. file [9].

- The server should be easy to debug. This is measured by the number of lines of code, the lowest the number of lines of code, the easier it is to debug the code [6].

4.3 Constraints:

- The server must be implemented in C programming language [8].
- The 'storage.h' file must not be modified [8].
- The server must be buildable by running the 'make server' command in the 'src' directory and produce an executable called 'server’ [8].
- The client library must build by running 'make libstorage.a' in the 'src' directory [8].
- The server and client implementations must pass all test suites provided [9].
- The server must satisfy the constraints set out in the 'storage.h' file e.g. Maximum table length, maximum key length, etc [10].

5 Design Decisions

We justify each of our design decisions below.

5.1 Data Structure

To suit our clients' specific situation and needs such as accommodating 150 million users concurrently and a more secure database, we implemented a hash-table and binary search tree combination. The hash-table will be used for insertion and search operations whereas the binary search tree (BST) will be used for the query operation. There are two sets of data to be stored: all of the tables, and all of the respective keys with their corresponding values in each table. For our client Adobe, the key would represent the username and its corresponding value would be the password. Since Adobe has a large volume of user data, in the magnitude of millions, the most important criteria to consider would be the speed of the database and its scalability with the amount of volume of data. Adobe users do not generally delete their user accounts because to delete a user account, one must go through the hassle of contacting customer services directly [14]. Therefore, the speed of deletion will not be considered. For the speed, three categories are considered; the speed of insertion, speed of searching for a particular entry, and the speed of query. Focus is drawn on how these speed changes as the amount of data increases. As listed in Table 1, the speeds may remain constant, increase logarithmically, or linearly as the size of data increases depending on the data structure used.

The most conventional data structures are listed in Table 1 with their evaluations according to our criteria. Only these three data structures were chosen because, at present, these data structures perform best in terms of speed. According to the data below, the hash table on average is the fastest in both insertion and deletion. As well, the space complexity, or how much memory is taken as the amount of entries scale up, is about the same for all three of the options considered below. Hence, we have chosen the hash table to store our keys and values, or the usernames and passwords.

Lastly, we have chosen to use the binary search tree as the storage method for tables. For Adobe's case, the tables may correspond to user information to for each of their different products. From our reasoning above, it would seem most viable to also use a hash table here as well as for the table values. However, it is found that a hash-table does not efficiently allow the user to print and list all of the table names, or query the entire database for specific dataset– which is a desired functionality. With this consideration in mind, binary search tree is the best choice. A binary search tree would overcome this limitation of the hash-table of not being able to query efficiently. It is also faster than linked list in terms of insertion and searching.

Table 1: Algorithms' Processing Speed [3]

Data Structures: (Large data sets)	**Processing Speed (Average):**			**Scalability (Space complexity):**
	insertion	search	query	
Hash Table	constant	constant	maximum*	linear
BST	logarithmic	logarithmic	linear	linear
Linked-list	constant	linear	linear	linear

* maximum because it always search through every available space in the database even if it is empty, which means it is the slowest. Therefore, it takes the maximum amount of time possible regardless of the size of data.

5.2 Adaptive Communication protocol

In the communication between the client and server, a serial form of data is transferred between the two in hardware [15]. For our implementation, we used a ‘+’-separated form of communication argument communication as it is not a valid character to be used in the arguments [10]. Alternatively, it was also possible to send each argument as its own small packet of data to the server one at a time. However, this alternative has several major drawbacks. First, it requires the client/server combination to process data to be sent and data to be received several times rather than just once – this decreases the efficiency of communication and would detract from our objective to maintain and efficient server database. Moreover, because data has to be sent multiple times, there is a much greater risk of a packet being lost thus making the entire sequence invalid. This potentially introduces more errors and would lead to a more inefficient and less robust database. As well, the situation is even more complicated to control and prevent from hack if there were multiple clients communicating with the server concurrently. Each client would be sending or receiving many more packets of data compared to having all of the data concatenated into one large packet and sent all at once.

5.3 Connection State

The client and server need to connect to each other every time communication between the two occurs. For the connection states between the client and the server, we chose to maintain an open connection between the client and server rather than opening and closing the connection each time the client makes a request. The main reason for this was that the alternative, which is ti make a new connection every time a request is made, is inefficient and slower. As well, the implementation on our part would be simpler and would hence reduce the implementation costs of the server. However, leaving the connection open leaves room for an attacker to hijack the connection and perform man-in-the-middle attacks. To prevent this, we made our implementation such that every time the client and server communicates, the data must be encrypted so that only the client and server can know the contents and verify the identity of the server and client each time [16]. This data encryption is automatically implemented within the storage functions and is not shown in Figure 3.

5.4 Error Handling

The code for storage acts as an interface between the client and the server [8]. It is also used to implement the client library as shown in Figure 3 [8]. Unfortunately, users tend to accidentally type invalid commands and requests to the database. We designed our code such that if there is an invalid command error on the client-side, such as having improper inputs, storage will return an error to the client and not make a request to the server to protect the server from behaving erroneously. Alternatively, we could let the server detect client-sided errors. However, by detecting the error in storage rather than server, we reduce the number of operations that needs to be taken before the error is detected. This improves efficiency and debugging. Unfortunately, the client library is unable to identify errors that need to be checked with the database as the database is only accessible by the server. Therefore, errors on the server-side are only going to be invalid request error from the client such as invalid table name or unauthenticated client. All of these invalid requests are passed back to storage which interprets the returned values into an error code. The use of error codes allows each error to be uniquely printed appropriately in human readable format for the client. This greatly improves the ease of which errors can be debugged in the database and would reduce the overhead costs of maintaining the database. As well, making the database easy to fix contributes to how robust it is.

5.5 Public Key Encryption

The encryption of data between two communicating users fall under two categories: symmetric key encryption and public-key encryption [5]. To encrypt our data, we have chosen to use the public key encryption protocol. Adobe's current encryption method uses the "symmetric key encryption scheme" which uses a single key to encrypt and decrypt [5]. Using this method, the user would first operate on data as needed and then encrypt it after they are done [17]. However, with this method, anyone with the original key can also subsequently decrypt any data that was encrypted using that key. Moreover, the fact that this key must be shared between any users that wish to communicate encrypted data increases the chances of the key getting leaked or stolen [17].

To address this problem, we implemented the "public key encryption" using the solitaire encryption algorithm by Bruce Schneier, which uses a pair of keys; one key is made public so that anyone may use it to encrypt their data, and one key is kept private so that only those with the private key may decrypt the data [5][29]. The major advantage of public encryption lies in the fact that users can encrypt their data without having to know how to decrypt. This way, only those that are supposed to receive the data, e.g. a central server, can decrypt the data.

Think of it as a mailbox, the public, who are Adobe's users are able to drop in their information such as their passwords and emails into the mailbox, but only Adobe who has the private key is able to open the mailbox and thus gain access to these sensitive information.

5.6 Parsing Algorithms

We chose to use LEX & YACC as it is most widely used and very robust, easy to use and update to produce parser functions for our configuration file but chose to use our own functions for everything else. The main reason for using LEX & YACC is because they can very powerful tools for writing robust parser functions; albeit it might take time to actually produce the parser functions. Since our configuration file parser needs a large overhaul due to the introduction of new constraints and rules to be fulfilled, we chose to rewrite our old parser completely using LEX & YACC. A highly robust parser has lower bugs, which reduces the loopholes where hackers can exploit. Also, an easy to update parser decreases costs for maintenance. Our parser functions for everything else however only needed a slight touch-up, so we opted to just build on and fix our old functions than to work from scratch. As well, we wrote the parser functions specifically to suit our database.

5.7 Compatibility issues

The introduction of new functions and rules for user inputs and arguments meant that a lot of code needed to be revamped. Due to the changes of how the configuration works, our server database is not backwards or forwards compatible. However, our client shell is still completely forwards compatible although it would not enable the use of new functions, only old ones as we adhere to our programmer objective embracing simplicity and only add new features as we need them [6].

5.8 Unit Testing

Our secure database code is fairly large as it contains 46 different functions. Unit testing enables us to test each function of our program in isolation [18]. Also, it allows continuous testing to ensure we did not introduce a new bug during the development process.

We focus on testing the program with constraints highlighted in the storage.h file such as the examples given in 5.3 above. To illustrate, we test that the database is initialize properly, that the basic functionalities of the server such as get, set, and query is working, and that it outputs the proper error messages to the client in the case of an error [10].

From the unit cases we have implemented, we have covered 65% (30 functions out of the 46 functions in total that is accounted for in our test program) of code coverage on codes in utils.c, storage.c, database.c, and server.c files. Functions from client.c file wasn't tested because we want to catch all our errors in the storage.c file as it is the client library that will be interfacing with many different clients whereas the client.c file is just one client by itself that is only used during code development.

5.9 Concurrency Mechanism

We focus on threading in order to support multiple clients connecting and accessing the server database at the same time. The various methods to maintain concurrency in the server are: threading, selects, processes. Primarily, our focus is on threading but the server can be configured to use the other two methods as well. The reason we focus on threading is because it has a clear advantage over the other two methods in terms of our objectives listed in section 4.2 [19]. In comparison, threading has the lowest complexity to code and to maintain concurrency and synchronization between the server and all the clients. Select is inefficient as it has one line of execution at a time when dealing with multiple clients and is complicated to code whereas processes is difficult to maintain synchronization because each client runs under a separate process, rather than under one single process as would be in threading. Since data is not shared between processes, it is harder to synchronize and allows hackers to exploit bugs that may be contained in the system. With this in mind, using threading would be the most robust method and is easy to implement, which leads to less overhead costs in the long run. Furthermore, threading is the fastest in terms of efficiency. Using separate processes per client means that each process has its own parameters which results in significant more overhead data; threading shares this overhead and so can save a lot of memory, which lowers the time needed to create duplicate new memories and thus is more efficient [20]. While threading shares some overhead, each thread still allocates a portion of memory for use in each thread. Select does not have to allocate any new memory and so would use the least memory out of the three options. However, due to the complicated coding nature of select discussed above, coding for select risks having inefficient code and a less robust database in general; therefore, threading is our preferred method of concurrency.

5.10 Data Synchronization

For threading, we implement critical sections to allow data synchronization is between clients. The most common mechanism to ensure synchronization is critical sections but other methods such as atomic operations exist as well. The efficiency between critical sections and other methods are comparable but generally differ greatly between different implementation of code [21]. As we cannot predetermine which method would be the most efficient for our implementation, our criteria for the best method of data synchronization would be ease of implementation. In this case, critical sections come out ahead as it is easy to use and implement [21]. Due to a simpler coding structure, our code would be easier to debug and more robust. As a result, using critical sections for our implementation can reduce overhead costs for maintaining the server.

5.11 Log Files Synchronization

Log files are used to keep track of server operations [22] and help to identify the cause of unclear malfunctions in the server, which aids in debugging [23]. Due to multithreading, log messages can be mixed up due to multiple logging functions accessing the log file at the same time [24]. Therefore, logging functions have to be executed atomically, meaning the operation completes in its entirety without interruption by other threads [25]. We use critical section to make our function atomic as a shared object between threads but can be acquired by one thread at a time and used to wrap around a specific code, in this case the logging function [25]. This section will prevent the function inside to be executed by the other threads while a thread is executing it [25] hence, making sure the logging messages are outputted correctly into the log file without mixing up. An alternative is using mutex, which operates similarly as critical [25] section, but has a lower operating speed is than critical section [26]. Although data synchronization is done by locking, log file synchronization are done atomically to allow proper ordering of output message.

5.12 Threads Management

Threads for our database was constrained to be created only when a client connects to the server and terminated only when the client closes the connection [27]. To fit this model, we use pthread_create to create a new thread whenever a connect command is received at the server. To delete, we chose for the thread's procedure to end whenever a disconnect command is received from the client. Alternatively, we could choose to end the thread at any other command from the client as well. However, this means that the thread might terminate before the client is done using the database and we would have to reinitialize a new thread if the client sends new commands. Terminating and starting a thread is inefficient, so we only terminated and initialize occasionally in order to maintain an efficient database.

5.13 Possible Attacks to Transaction Processing

Malicious client application may change the record metadata by setting a higher counter for the record. The higher counter means that the record is at the most updated version in multithreading and will be stored into the database, which results in future clients accessing the wrongly updated data. We prevent this by only allowing actual users to change the record counter to 1 above its previous counter. This means that malicious client application will need to first access the data before it is able to change it. As show in Appendix B, access is only granted to a user who logs in correctly and malicious client applications are unable to do so.

6 Conclusion

Our database is specialized for Adobe's needs. It contains the functionality and security to store highly confidential customer information. Our server database is scalable and runs efficiently for all 150 million adobe users. We chose hash table based as it is known for its high speed in searching and inserting customer information. Our simple database will be less susceptible to errors compared to Adobe's current complex database. It also cost less while still providing the same functionality that Adobe requires. Finally, our secure database has highly advanced encryption algorithm to meet Adobe's needs for a secure database.

7 References

[1] A. Levin. (2013, Nov 17). *Is the Adobe Hack Just the Tip of the Iceberg?* [Online]. Available: http://gma.yahoo.com/adobe-hack-just-tip-iceberg-120449810--abc-news-topstories.html

[2] A. Hern. (2013, Nov 7). *Did your Adobe password leak? Now you and 150m others can check.* [Online]. Available: http://www.theguardian.com/technology/2013/nov/07/adobepassword-leak-can-check

[3] T. H. Corment, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* MA: MIT press, 1990

[4] Adobe, Working with local SQL databases in AIR. [Online] Accessed February 9, 2014. Available: http://help.adobe.com/en_US/as3/dev/WS5b3ccc516d4fbf351e63e3d118676a5497-7fb4.html

[5] J. Tyson. *How Encryption Works.* [Online]. Available: http://computer.howstuffworks.com/encryption3.htm

[6] E. Raymond, “Chapter 13. Complexity,” in The Art of Unix Programming, MA: Addison Wesley. [E-book] Available: FAQS.ORG

[7] “M3 Use Case Scenario,” [Online] Available: https://docs.google.com/file/d/0B7LwMIcTMJGobjd5X0NxRFh1eGM/edit

[8] *Detailed Specification* (n.d.) [Online]. Available: https://sites.google.com/a/msrg.utoronto.ca/ece297/assignment-1/detailed-specifications

[9] *Detailed Specification* (n.d.) [Online]. Available: htps://sites.google.com/a/msrg.utoronto.ca/ece297/assignment-2/detailed-specifications#TOC-Workload

[10] *Detailed Specification* (n.d.) [Online]. Available: https://sites.google.com/a/msrg.utoronto.ca/ece297/assignment-3/detailed-specifications

[11] Microsoft, SQL Enterprise Edition. [Online] Accessed March 6, 2014. Available: http://www.microsoft.com/en-us/sqlserver/editions/2012-editions/enterprise.aspx

[12] J. Nielsen. (2001, Jan. 21). *Usability Metrics.* [Online]. Available: http://www.nngroup.com/articles/usability-metrics/

[13] J. Morris (1998). *Data structures and Algorithms.* [Online]. Available: https://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html

[14] Adobe, Delete your Adobe ID account. [Online] Accessed February 9, 2014. Available: https://wikidocs.adobe.com/wiki/display/wel/Delete+your+account

[15] "4. Serial Communication," [Online] Available: http://claymore.engineer.gvsu.edu/~jackh/books/plcs/chapters/plc_serial.pdf

[16] *SSH Man-in-the-Middle Attack and Public-Key Authentication Method (*2010, Dec. 25) [Online]. Available: http://www.gremwell.com/ssh-mitm-public-key-authentication

[17] R. Kayne. *What is an Encryption Key?* [Online]. Available: http://www.wisegeek.org/what-is-an-encryption-key.htm

[18] *Unit testing with Check* (n.d.) [Online]. Available: https://sites.google.com/a/msrg.utoronto.ca/ece297/tas-and-labs/unit-testing-with-check

[19] *Pthreads VS select* June 4, 2004. http://www.linuxquestions.org/questions/programming-9/pthread-vs-select-166708/

[20] *Concurrent connections with threads* (n.d.) [Online]. Available: https://sites.google.com/a/msrg.utoronto.ca/ece297/tas-and-labs/concurrent-connections-with-threads

[21] Apple (2013, October), Threading Programming Guide. [Online]. Accessed March 29, 2014. Available: https://developer.apple.com/library/mac/documentation/cocoa/conceptual/Multithreading/ThreadSafety/ThreadSafety.html

[22] Brick Marketing, What is a Log File? [Online]. Accessed March 29, 2014. Available: http://www.brickmarketing.com/define-log-file.htm

[23] IBM (2007, August), Logging in multi-threaded applications efficiently with ring buffer. [Online]. Accessed March 29, 2014. Available: https://www.ibm.com/developerworks/aix/library/au-buffer/

[24] *Logging in a Multithreaded Environment* (2012, Dec 17) [Online]. Available: http://blog.instance-factory.com/?p=181

[25] K. Coe (2013, May 17). *Introduction to Multi-Threaded, Multi-Core and Parallel Programming concepts.* [Online]. Available: http://katyscode.wordpress.com/2013/05/17/introduction-to-multi-threaded-multi-core-and-parallel-programming-concepts/

[26] Intel (2011, March), Choosing Between Synchronization Primitives. [Online]. Accessed March 29, 2014. Available: http://software.intel.com/en-us/articles/choosing-between-synchronization-primitives

[27] *Detailed Specification* (n.d.) [Online]. Available: https://sites.google.com/a/msrg.utoronto.ca/ece297/assignment-4/detailed-specifications

[28] P. Kimmel, *UML Demystified*, Emeryville: McGraw-Hill, 2005.

[29] B. Schneier. (1999, May 26). *The Solitaire Encryption Method.* [Online]. Available: https://www.schneier.com/solitaire.html

8 Appendices

A Written Use Case

- Primary Actor:
  - Adobe Client: The client will be interested in ordering new products offered by Adobe and will be able to track their product upon ordering.
- Level:
  - Summary Level System: This is a multiple level system as it benefits both Adobe's client and Adobe's manager.
    - Adobe's client can order and track their ordered products.
    - Adobe's manager can find out the market segmentation of its customers to be able to broadcast products that may interest a certain segment of their market.
- Stakeholders and Interests:
  - Adobe's Client
    - To have secure purchase and ability to track their ordered products.
  - Adobe's Managers
    - To be able to market their products to interested customers by analyzing its market using market segmentation.
  - Hackers
    - To realize the system is robust and be unable to hack into the system to gain customer's sensitive information such as credit card numbers, emails, and passwords.
  - Victims of November 2013 Adobe Hack
    - To be able to change their passwords securely.
  - Financial Institutions
    - The payment details of their customers who buys Adobe's product is kept private and confidential from public but also shared with the financial institution.

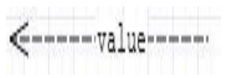
        - To ensure that their customer’s card is verified and used only by that customer and not a hacker who stole the customer’s credit card details.

- Precondition:
    - Sign Up
        - Every user must have an adobe account before they are able to access the system.

- Minimal Guarantee:
    - Logging In
        - Every user must logged in to their existing account with the right password and email information before being able to use the system.
- Success Guarantee:
    - The user is able to track his product upon ordering.
- Main Success Scenario and Extensions:
    - 1. Adobe’s user or Adobe’s manager logs in to their existing adobe account with the right password and email information.
    - 1.1. Adobe’s user orders a new product that is offered by Adobe.
    - 1.1.1 Adobe’s user can view order information, change their user’s information such as password, email, shipping information, or credit card details upon ordering.
    - 1.1.1.1 Adobe’s user will be given tracking information upon successful order.
    - 1.1.1.1.1. Adobe’s user can now track their product for information.
    - 1.1.2. The specific product ordered is out of stock and cannot be ordered. The error is reported to the Adobe’s user.
    - 1.2. Adobe’s user can track any previously ordered product.
    - 1.3. Adobe’s manager checks its market segmentation by querying for users based on a specific interest, age, previously bought products, or geographic location.
    - 1.3.1 The system returns Adobe’s manager a list of users with their information such as email, purchase history, age, and geographical location that matches the query.
    - 1.3.1.1 Adobe’s manager can broadcast information to these users via email.

  - 1.4 Adobe’s manager can immediately broadcast information to their users by including emails.

B UML Diagrams Specifics

- UML Case Diagram: [28]
  - Pre-Condition : <<include>>
  - Extensions: <<extend>>
  - Error extensions: <<error>>
- UML Sequence Diagram: [28]
  - $\longleftarrow$: An unknown incoming source.
  - 

: Returning a value from a given request.