EXTENDED STORAGE SERVER FOR APARTMENT BUILDINGS

MARCH 8, 2014

Aryamman Jain, Pranav Mehndiratta & Vaibhav Vijay
Team ID: CD-037

Contents

Executive Summary

Apartment buildings generate large amounts of data on a daily basis. This data needs to be recorded to assure smooth building operations and ensure residents' safety. Our project focuses on providing a centralized and fast storage system with a client-server interface to replace the current obsolete paper-based system.

Our storage server reduces the time required for building personnel to access resident details and facilitate communication among the landlord, management staff and tenant. Moreover, it will enhance the security of the data storage medium by using encryption methods and multi-layer communication protocols. The system handles various possible errors and effectively logs the activities of the client-server application.

Apart from serving the management personnel, the system will also benefit the tenants. This project will help the management track tenants' rent payments, check their lease status and respond quickly to maintenance requests. Tenants can also be notified immediately upon receiving any mail/packages.

The backend of the system is optimized to provide rapid access and modification capabilities of the database records. This high speed is a result of a refined database architecture for storing tenant details (contact info, lease agreement, rent status and more). The speed does not compromise the flexibility or robustness of the storage server, as it is not limited to handling residential data. Other storage media are vulnerable; our storage server only allows access to users with valid credentials.

Resulting from our design decisions, the development of this project is not as complex as other database systems; thus, reducing the cost of ownership. Time saved through the intuitive and accurate implementation of this system can be diverted to improving the residential experience at the apartment building. We aspire to fabricate a state-of-the-art database system that will set the industry standard in the near future.

Introduction

For our storage server project, we are implementing a database for an apartment building in Downtown Toronto. This document's focus is to provide the reader with an overview of the client requirements and the design's ability to fulfil them. Our primary goal is to cater the needs of the management office by providing them with a centralized database. The end users would include the authoritative staff of the property; specifically, the superintendent and the security department.

Residential buildings generate vast amounts of data every day that need to be tracked for security and managerial purposes. This data includes tenant details, rent status, agreements, maintenance requests, and pending notices for residents [1]. These details can be collected from the resident upon their moving into the building [2]. Secure and efficient storage of this data is imperative to provide a decent communication bandwidth between the tenant, management staff and landlord.

The current system lacks the necessary automation of record keeping as it is entirely paper based. The prime duty of security guards, at the building in question, is to record and store a variety of data in large log books kept at his desk [3]. The management office asserted that the only method to contact the tenants is by dropping letters at each apartment [1]. Whereas, there is no form of direct communication between the superintendent and tenant, except for coincidental meetings [4].

Our proposed system will implement a centralized server that will store records in a particular data structure. This database can be accessed and modified by the client through a given interface that is capable of sending requests to the server using a specific protocol. To maintain the privacy of this data, server access is restricted through the use of login credentials that are configured as the server connection is established. The server itself may also internally store the database in an encrypted form. The client will be able to run queries on this database for tasks such as communicating with residents and securely obtaining resident(s) information.
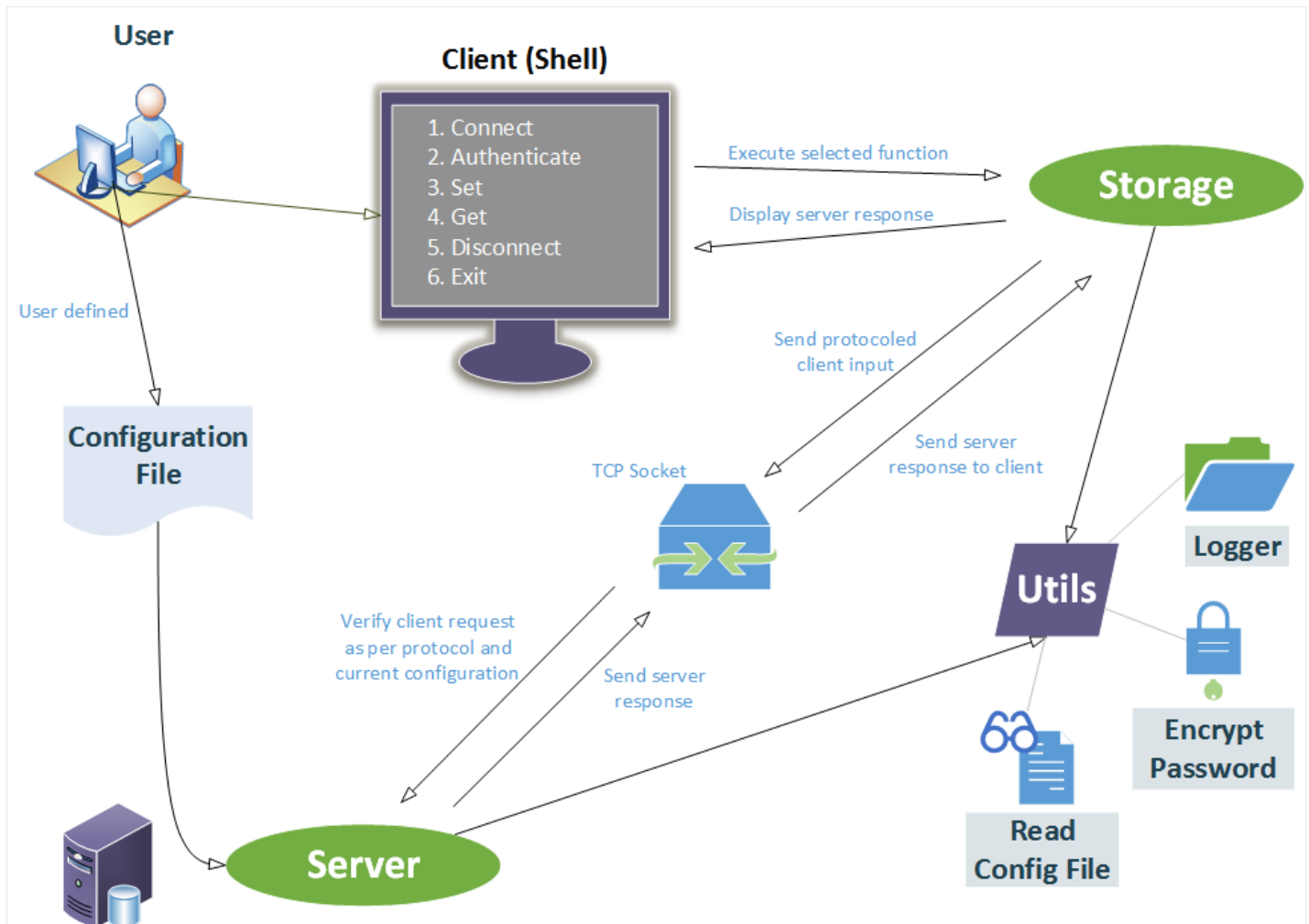
Software Architecture - UML Diagrams

Figure 1: UML Component Diagram

When the server is started, user-defined configuration file is read using a function defined in the utilities. The client shell is the medium for intercommunication between the user and client library (storage). The communication between the client library and server follows a definite communication protocol. This client library consists of functions that are called based on user input. The server then responds with a pass/fail along with a value (if applicable) and an error/valid output is displayed on the shell. Components:

- Client (shell) – Display seen directly by the user
- Storage – Client library containing functions to communicate with the sever
- Server – Database is stored here
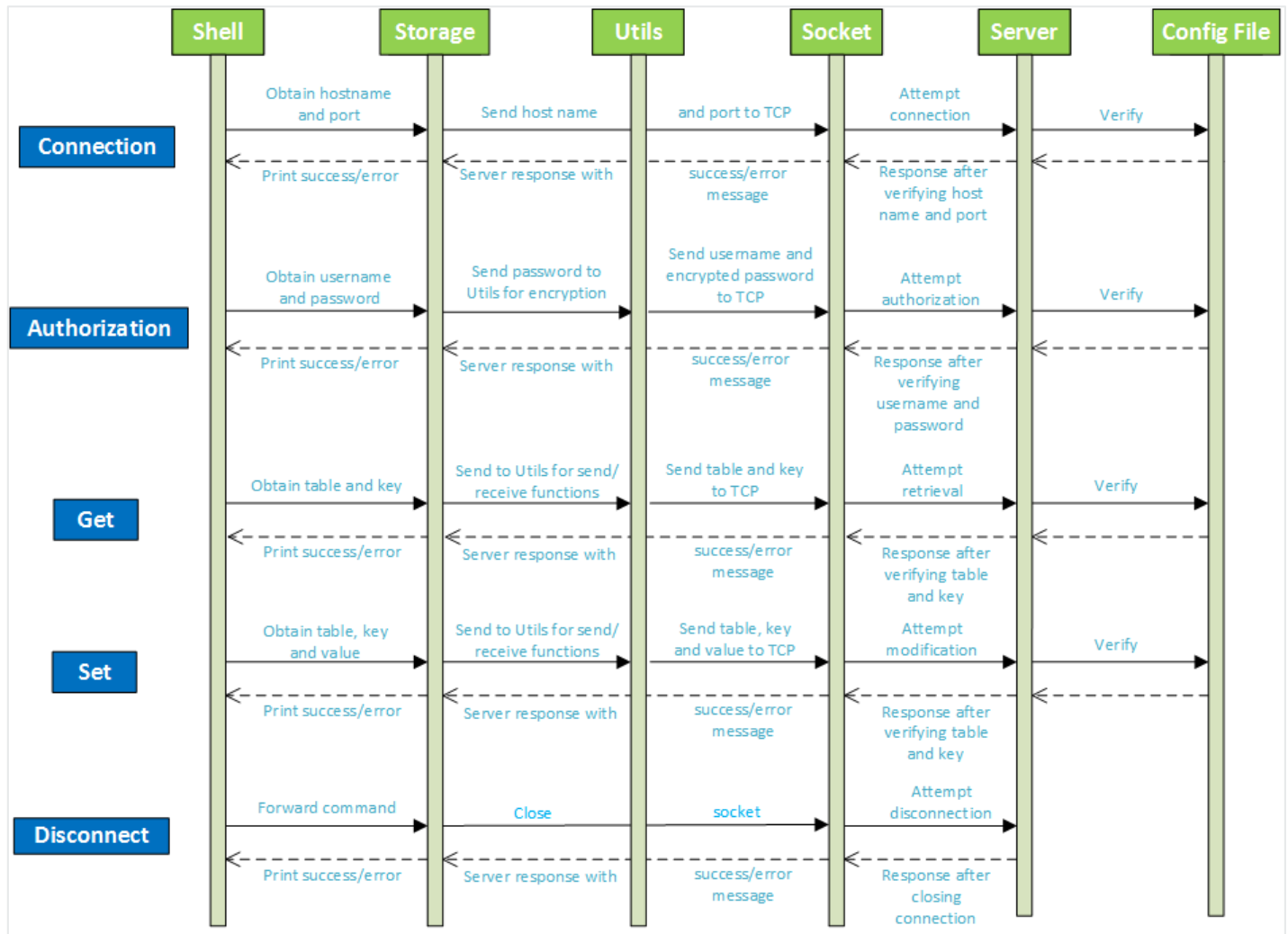- Utilities – File containing functions common to entire project

Figure 2: UML Sequence Diagram

- **Connection**: User inputs hostname and port which are then sent to the server to start a connection.
- **Authentication**: User inputs username and password which are sent to the server for verification.
- Get: User inputs 'table' and 'key' which are sent to the server database to attempt retrieval. Value at position 'key' in table 'table' is returned if no errors occur.
- **Set**: User inputs 'table', 'key' and 'value' which are sent to the server database to attempt modification. Value at position 'key' in table 'table' is modified to 'value'.
- **Disconnect**: User selects disconnect on client shell and the socket (if open) is closed.
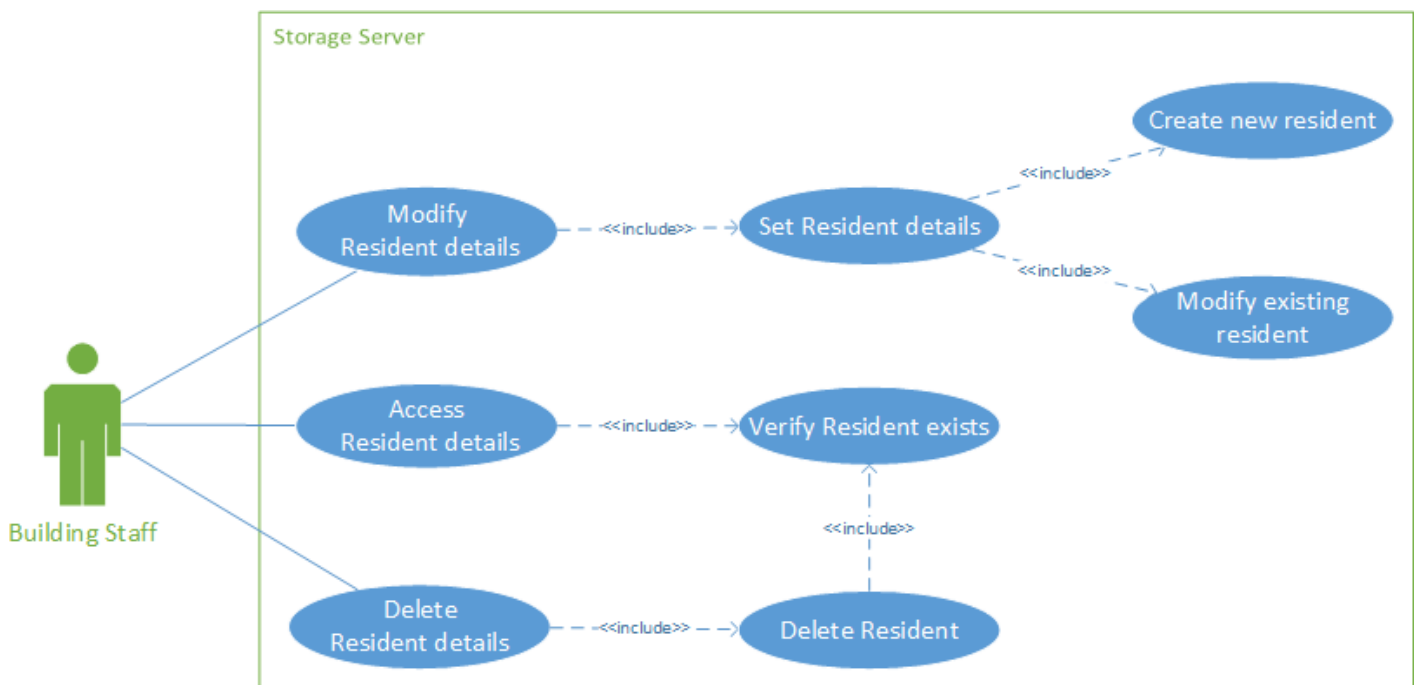- **Error code**: set and printed if process fails at any step.

Figure 3: Use Case Diagram

- **Primary Actor**: Building management office, superintendent, security guards
- **Level**: User level – system only benefits the primary actor
- **Stakeholders**:
  - *Building Staff (Client)*: The resident administration who will use the storage server
  - *Server Developers*: The programmers who maintain the storage server
  - *Building Residents*: Tenants whose details will be recorded in the storage server
- **Precondition**: Client has connected and authorized successfully with the server using valid credentials as set up in the configuration file
- **Minimal Guarantee**: Encryption of the client's login details to facilitate proper authorization, logging connection activities and errors
- **Success Guarantee**: User is authorized successfully, and accesses/modifies/deletes an existing resident entry in the database
- **Main Success Scenario**:
  1. User connects and attempts to authorize
  2. Server verifies user login credentials
  3. Server requests next command (add/modify/delete resident records)
  4. User enters desired command
  5. Server verifies command request and asks for table name, key and value accordingly
  6. Server responds to commands entered after verifying the given arguments

- **Extensions**:

  4a. User enters invalid command

  4b. Server prompts user to enter another command

  6a. User enters non-existent table name

  6b. Server notifies user that table does not exist

  6c. User enters non-existent key

  6d. Server notifies user that key does not exist

System Requirements

The following are the major requirements that the storage server must fulfil which have been developed as per the client's desired attributes of the storage server [1] [3].

FUNCTIONS

These are the major functionalities required of the storage server to facilitate client-server communication and data storage.

- Centralize all data to one location and provide multi user connections
- Provide a stable connection to the server (`storage_connect`)
- Authenticate client against valid username and password defined (`storage_auth`)
- Retrieve multiple records based on defined predicates (`storage_query`)
- Retrieve an existing resident record (`storage_get`)
- Insert/Update/Delete a resident record (`storage_set`)
- Disconnect from the server safely (`storage_disconnect`)
- Parse server configuration file set by user and set-up server accordingly (`read_config`)
- Reduce amount of paper work that needs taking care of
- Log all client-server interactions in time-logged files

OBJECTIVES

These are the goals that the functionalities should aim to fulfil and satisfy client requirements.

- Implement the shell and protocol to allow client to communicate with the storage server
- Implement a communication protocol between server and client (send/receive through socket)
- Ensure server doesn't prematurely respond with success messages
- Manage tenant contact details and facilitate rapid access (time needed to obtain data)
- Possess an appropriate data structure for multiple record insertion in a hierarchical manner (add/change data to the table is not complicated)
- Provide means of effective communication among landlord, management and tenant using proper querying commands
- Facilitate debugging of client-server operations with proper error messages and logging history (log all success/error codes to `stdout` /file on client and server sides)

SYSTEM REQUIREMENTS

CONSTRAINTS

The following constraints must be met by the storage server as they are required by both the client and the programmers of the client-server.

- Client-based
    - Shall not use any personal details without tenant's permission
    - Server shall have enough memory to store the entire building's details
    - Access shall be restricted to building personnel
- Technical-based
    - Code must be written in C [5]
    - Client interface file 'storage.h' shall not be changed [5]
    - Server shall respond to client requests running on separate machines
    - Server shall only accept configuration files of given format (Appendix A)
    - Client-server communication is governed by Transmission Control Protocol (TCP) [6]

CRITERIA

To successfully meet the above outlined objectives, the designer may use these metrics to evaluate the storage server.

- Performance of database in sending and receiving data (Metric: time needed to store and access records)
- Reliability of stored data (Metric: data consistency following modification of records)
- Intuitiveness of client-server interface (Metric: time and support required to teach usage of new software)

Design Decisions

DATA STRUCTURE

To allow the client to access each resident record efficiently and also maintain consistency on the server side, the data structure must be robust and allow for quick insertion/deletion/searching of records. Various data structures are available to be used for these purposes such as arrays, linked lists, trees, and hash tables. A full comparison is shown in Table 1 based on order of time complexity. This complexity increases with the number of data records stored [7], thus enforcing a limit on the number of residents the user can store in the system. Hence, having a lower order of complexity allows for more data capable of being stored. Table 2 shows the various advantages and disadvantages of different data structures. Since speed is a major factor to the management [1], hash tables would be the ideal data structure to store each table. The limitless capacity of hash tables and their performance consistency independent of number of records would best suit our client's needs.

Data Structure	Complexity			
	Insert	*Delete*	*Search*	*Space usage*
Array	O(n)	O(n)	O(n)	O(n)
Linked List	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(n)
Hash Table	O(1)	O(1)	O(1)	O(n)

Table 1: Data Structures Comparison [7]

Data Structure	Advantages	Disadvantages
Array	Fast indexing and iteration	Wasted space if array is not full
	Easy to implement	Indices are fixed
Linked List	List can be traversed from either direction	Entire list needs to be traversed to find a record
	No limit on number of records	Complexity is high since data is not indexed
Binary Search Tree	No limit on number of records	Hard to implement
	Elements are always sorted	Constant complexity of O(log n)
Hash Table	Fastest access to data elements	Need to define a hash function
	Constant complexity of O(1)	Data is not sorted
	Minimal collisions in residential data	

Table 2: Comparison of Data Structures [7]

PARSING

Configuration File

Processing the configuration file involves extracting the connection information and authentication details. The extended configuration involves the database schema including table names, column

names that are part of each table and the type of data (integer/characters) each column accepts [5]. A sample of the defined format for the server configuration file is given in Appendix A.

Data

Each key-value pair in a table contains columns, i.e. multiple values per key. These are represented as comma separated strings with the column name followed by the actual value [5]. User input is converted into the protocol that our server understands (Table 3): #table contains the table name, #key contains the key, and #value would contain all the columns as a comma separated string. We will be using Lex and YACC to implement our parsing techniques in order to process a more complex table schema. This method of parsing is more robust and also less prone to create errors, as these parsing techniques are more developed [8]. The client will experience less errors as their input will be properly parsed and any invalid input will not be communicated to the server.

CLIENT-SERVER COMMUNICATION PROTOCOL

As communication between the client and server is governed by TCP [9], the protocol should minimize the bandwidth between the client and the server. The sending and receiving of requests/responses costs the majority of the time when the user interacts with the server. The client would like the quickest response time of the server as possible. Thus a streamlined protocol such as the one exemplified in Table 3 ensures only the necessary data is communicated between the client and server. This shall minimize the time cost of the communication, hence the response time of the server.

Function	Protocol*	
	Client Request	Server Response
`storage_auth`	`AUTH #username #enc_password`	`AUTH #pass` `AUTH #fail`

`storage_query`	`QUERY #table #predicates`	`QUERY`
		`QUERY #table`
		`QUERY #table #keys`

`storage_get`	`GET #table #key`	`GET`
		`GET #table`
		`GET #table #key #value`

`storage_set`	`SET #table #key #value`	`SET`
		`SET #table`
		`SET #table #key #value`

	*All strings are terminated by a new line ('\n') character	

Table 3: Client Server Communication Protocol (Rev 2)

CODE CHANGES & COMPATIBILITY

The new format of values breaks our code on the server side as we have to introduce new parsing methods and change the protocol. Hence, our server is not backward compatible. A new function for querying the database (`storage_query`) has also been introduced. Our client side code is not broken because it only requires the addition of this function. Hence, our client is backward compatible.

DESIGN DECISIONS

ERROR HANDLING

The client should be aware of any errors they may encounter as they use the storage server, therefore numerous errors are handled and logged throughout the client-server interaction. For each of these errors, a specific error code is flagged and logged, also displaying it to the user as the error is encountered. On experiencing errors while attempting to connect the client and server due to invalid connection information, or invalid login credentials, the server disconnects and ends the session. Whereas, errors due to accessing/modifying a record are flagged and returned to the client, but the server only prompts the user to enter a valid record. The variety of errors handled will allow the client to report any shortcomings of the system, and also help them understand how to use it efficiently at the same time. For detailed error codes and their occurrences, refer to Appendix A.

STATE OF THE CONNECTION

To ease interaction between the client and the storage server, once the connection is successfully established with proper authorization, it is reused for multiple get/set requests. To track the connection state, we describe it using three attributes; connection status, authorization status and a socket file descriptor. The server disconnects only when requested to do so, or upon any error with respect to establishing a connection (invalid configuration file or credentials). If multiple clients attempt to connect to the server simultaneously, with the same server configuration, the server will reject the connection. Clients may only connect simultaneously to the server using a different port.

TEST CASES

These test cases cover major client library functions listed below. They facilitate in testing the parsing implemented in these functions and our defined protocol.

Function	Test Case				
All	Input non-existent table	Input special characters/spaces in table name/key		Running command before connecting and authenticating	
`storage_get`	Input non-existent key				
`storage_set`	Input non-existent key (No Error)	Input special characters in value	Input spaces in value (No Error)	Input incorrect pair of column and value	Delete a non-existent key
`storage_query`	Input non-existent predicate	Input multiple predicates per column	Input predicate with missing comparators	Compare mismatching data types	Using mismatching data types and comparators

Table 4: Test Cases

Conclusion

This document has highlighted the primary components and architecture of an electronic system that will digitize data generated at residential buildings. The main goal of this storage server is to provide a secure, accessible and efficient method to store essential data. The client's requirements are iteratively assessed against the system's present functionality to best meet the necessary implementation. This is essentially following the Agile Development Model while developing this storage server.

The storage system design is formed by the decisions taken in this document; however, it only represents the current state of the design and will reflect any future changes in the client's requirements throughout the development process. For the time being, the major implementations are the hash table data structure, client-server protocol, error handling, parsing of config file/data, changes to be made in our code, and test cases.

Using this approach for record keeping at residential buildings, we hope to facilitate managerial tasks such as tracking a tenant's rent payments, maintenance requests and effective communication between different entities.

Bibliography

[1] A. Williams, Interviewee, *Detailled Discussion with the Management Personnel.* [Interview]. 3 February 2014.

[2] A.V. Powell & Associates LLC, "Forecast - A.V. Powell & Associates LLC," 2003. [Online]. Available: http://www.avpowell.com/wp-content/uploads/2012/08/Resident-Demographic-forms-and-instructions-website-version-6-16-03.pdf. [Accessed 8 March 2014].

[3] M. R. Ravi, Interviewee, *Initial Interaction with Client Personnel.* [Interview]. 2 February 2014.

[4] V. Hacuman, Interviewee, *Further Research with Authoritative Staff.* [Interview]. 2 February 2014.

[5] ECE 297, "Milestone 2 - Detailed Specification," 2014. [Online]. Available: https://sites.google.com/a/msrg.utoronto.ca/ece297/assignment-2/detailed-specifications. [Accessed 8 March 2014].

[6] RFC Sourcebook, "TCP, Transmission Control Protocol," 2012. [Online]. Available: http://www.networksorcery.com/enp/protocol/tcp.htm. [Accessed February 2014].

[7] C. E. L. R. L. R. a. C. S. Thomas H. Cormen, "Introduction to Algorithms," MIT Press, 2009, pp. 253-285.

[8] "Lex and YACC primer/HOWTO," 20 September 2004. [Online]. Available: http://ds9a.nl/lex-yacc/cvs/lex-yacc-howto.html. [Accessed 9 March 2014].

[9] B. ". J. Hall, "Beej's Guide to Network Programming: Using Internet Sockets," 3 July 2012. [Online]. Available: http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html. [Accessed 9 March 2014].

APPENDICES

Appendices

APPENDIX A – DESIGN DECISION SAMPLES

Name	Value
server_host	A string that represents the IP address (e.g., 127.0.0.1) or hostname (e.g., localhost) of the server.
server_port	An integer that represents the TCP port the server listens on.
username	A string that represents the only valid username for clients to access the storage server.
password	A string that represents the encrypted password for the username.
table	A string that represents the name of a table. There may be more than one table parameter, one for each table.
column	A string that represents the name of the column in the preceding table.
value_type	A string that represent the name of the type of the value for the column as the preceding column name. (int or char[SIZE], where SIZE represents the maximum length of the string)

Table 5: Configuration file parameters [5]

Duplicates of any parameter are unaccepted and will result in an error.

Correct Configuration	server_host localhost server_port 1111 username admin password xxxnq.BMCifhU table subwayLines name:char[30],stops:int,kilometres:int
Wrong Configuration	server_host localhost server_port 1111 server_port 2222 username admin password xxxnq.BMCifhU table subwayLines name:char[30],stops:int,kilometres:int
Wrong Configuration	server_host localhost server_port 1111 table subwayLines name:char[30], stops:int, kilometres:int table subwayLines stops:int, kilometres:int

Table 6: Sample Configuration Files [5]

Error Code	Meaning
ERR_INVALID_PARAM	This error may occur in any of the five functions if one or more parameters to the function does not conform to the specification.
ERR_CONNECTION_FAIL	This error may occur in any of the five functions if they are not able to connect to the server.
ERR_AUTHENTICATION_FAILED	This error may occur if the client provides wrong username and password to `storage_auth`.
ERR_NOT_AUTHENTICATED	This error occurs if the client invokes `storage_get` or `storage_set` without having successfully authenticated its connection to the server.
ERR_TABLE_NOT_FOUND	This error may occur in the `storage_get` or `storage_set` functions if the server indicates that specified table does not exist.
ERR_KEY_NOT_FOUND	This error may occur in the `storage_get` function if the server indicates that the specified key does not exist in the specified table. It is also used when trying to delete a non-existing key.

Table 7: Error Codes [5]

APPENDICES

APPENDIX B – SECTION WISE AUTHOR/REVISION/PROOFREADING

Contributor	Name	Student Number
AJ	Aryamman Jain	999554076
PM	Pranav Mehndiratta	999480725
VV	Vaibhav Vijay	1000073029

Section	Authored	Revision	Proofread
Executive Summary	AJ PM	AJ PM VV	AJ PM VV
Introduction	AJ PM VV	AJ PM VV	PM VV
System Architecture – UML Diagrams	PM	PM VV	AJ PM
System Requirements	AJ PM VV	AJ PM VV	AJ VV
Design Decisions	AJ PM	AJ PM VV	AJ PM VV
Conclusion	AJ PM	AJ	AJ PM VV