SCALABLE AND CONCURRENT SERVER FOR APARTMENT BUILDINGS

MARCH 30, 2014

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

Contents

Executive Summary 1
Introduction 2
Software Architecture 3
UML Component Diagram 3
UML Sequence Diagram 4
Use Case Diagram 5
System Requirements 6
Functions 6
Objectives 6
Criteria 7
Constraints 7
Design Decisions 8
Concurrency 8
Threads vs. Processes 8
Synchronization – Mutex vs. Semaphore 9
Logging 10
Implementation 10
Data Structure 11
Parsing 11
Configuration File 11
Data 12
Client-Server Communication Protocol 12
Error Handling 13
State of the connection 13
Testing 13
Conclusion 14
Works Cited 15

TABLE OF CONTENTS

Appendices _____ 17

Appendix A – Configuration File _____ 17

Appendix B – Data Structures _____ 18

Appendix C – Error Codes _____ 19

Appendix D – Test Cases _____ 20

Appendix E – Section Wise Author/Revision/Proofreading _____ 21

EXECUTIVE SUMMARY

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 interaction.

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 their 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. The system implements an intuitive user interface that allows inserting new data, modifying or removing existing data, and querying (searching) the database for specific records according to user-specified conditions. These functions can be performed by multiple users concurrently; the backend will handle conflicts in commands and maintain the consistency of the database. The high speed is a result of a refined database architecture for storing tenant details (contact info, lease agreement, rent status and more). The storage server is robust in its ability to process different data forms, and is not restricted residential details. This flexibility reduces the access and modification speed of the system; however, our application is able to minimize this negative effect and provide an optimal balance between speed and versatility. Conventional storage media (papers, outdated server systems, etc.) are vulnerable to unauthorized access and malicious users that want to access or modify the recorded data; however, our storage server only allows access to the database to users that possess the 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.

INTRODUCTION

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 provide the functionality required by the management office by providing them with a centralized database. The end users are building officials at 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]. Tenant details can be collected from the resident upon their moving into the building and other information can be entered into the database as per the building management's required usage [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, because 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 their desk [3]. The management office asserted that the only method to contact the tenants is by dropping letters at each apartment [1]; moreover, the superintendent has no form of direct communication with the tenants, except for coincidental meetings [4].

Our proposed system will implement a centralized server that will store records in a hash table data structure. This database can be accessed and modified by the client through a shell designed to send requests to the server using a protocol described in the design decisions that focuses on minimizing communication between the client and server. 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 COMPONENT DIAGRAM

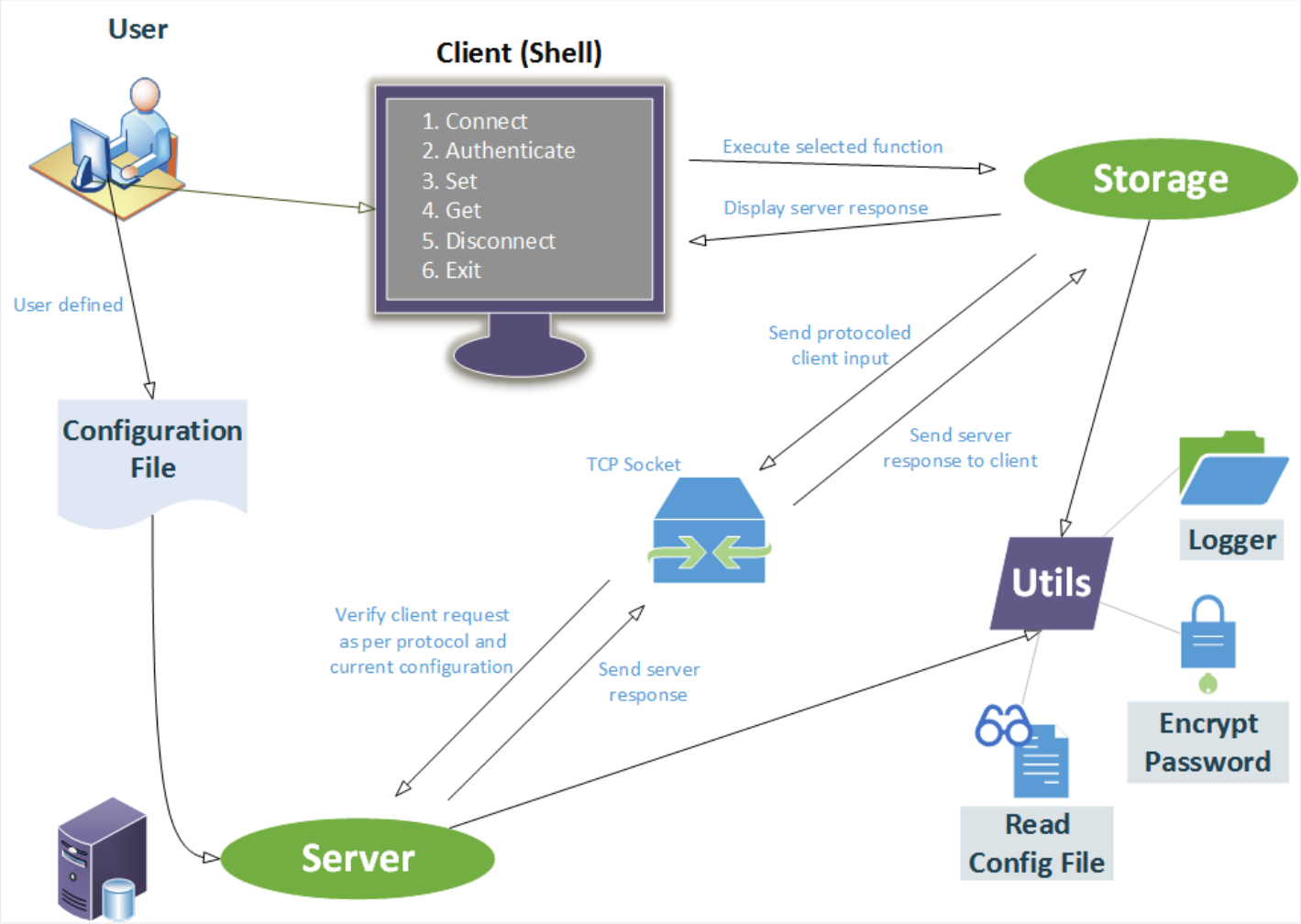

Figure 1: UML Component Diagram - Depicts how different components are linked together in the system

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

SOFTWARE ARCHITECTURE

UML SEQUENCE DIAGRAM

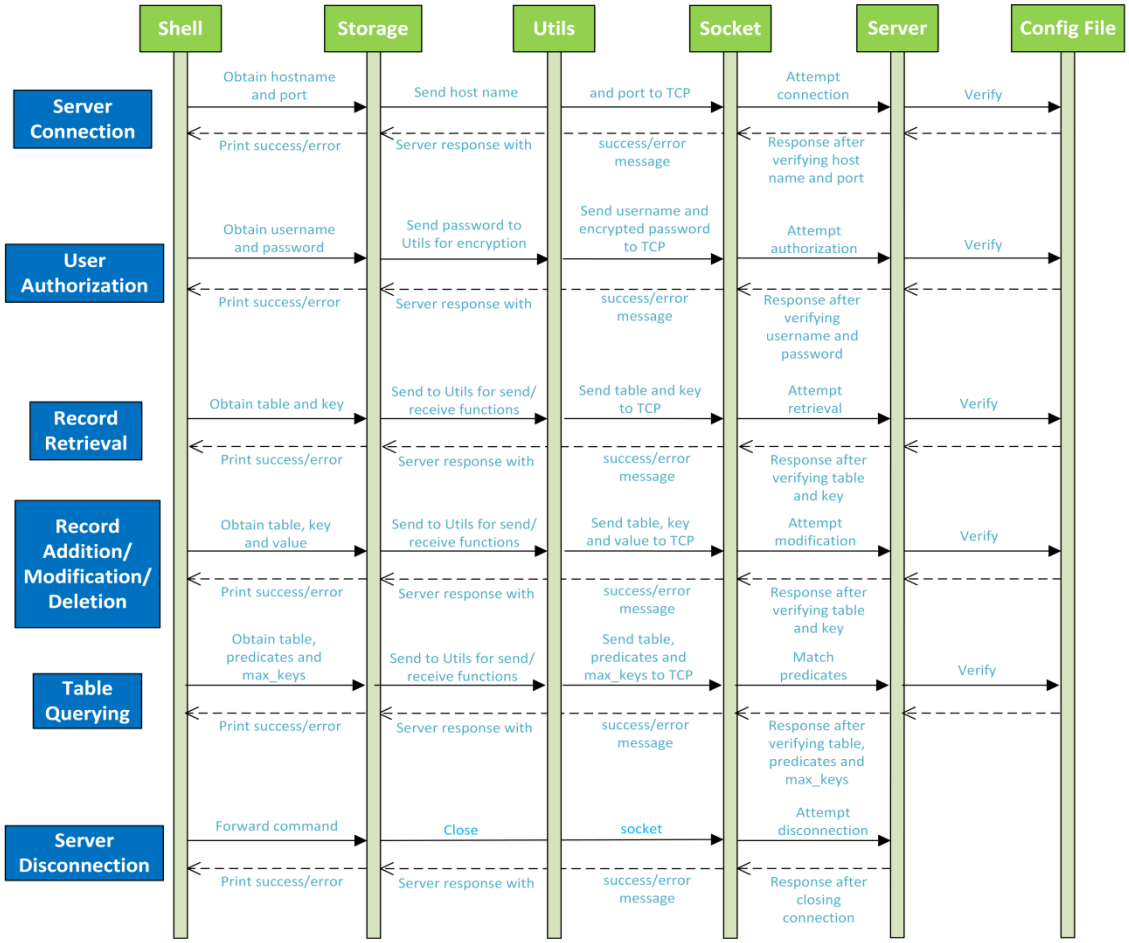

Figure 2: UML Sequence Diagram - Depicts how different processes operate with one another and in what order (shown by arrows)

- **Server Connection** (`storage_connect`): User inputs hostname and port which are then sent to the server to start a connection.
- **User Authentication** (`storage_auth`): User inputs username and password which are sent to the server for verification.
- **Record Retrieval** (`storage_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.
- **Record Addition/Modification/Deletion** (`storage_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'.
- **Table Querying** (`storage_query`): User inputs 'table', 'predicates' and 'max_keys' which are sent to the server database to attempt retrieval of keys for records that satisfy the user specified predicates. The number of matched keys is returned, but not more than 'max_keys' keys in the table 'table' that match 'predicates' are returned.
- **Server Disconnection** (`storage_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

SOFTWARE ARCHITECTURE

USE CASE DIAGRAM

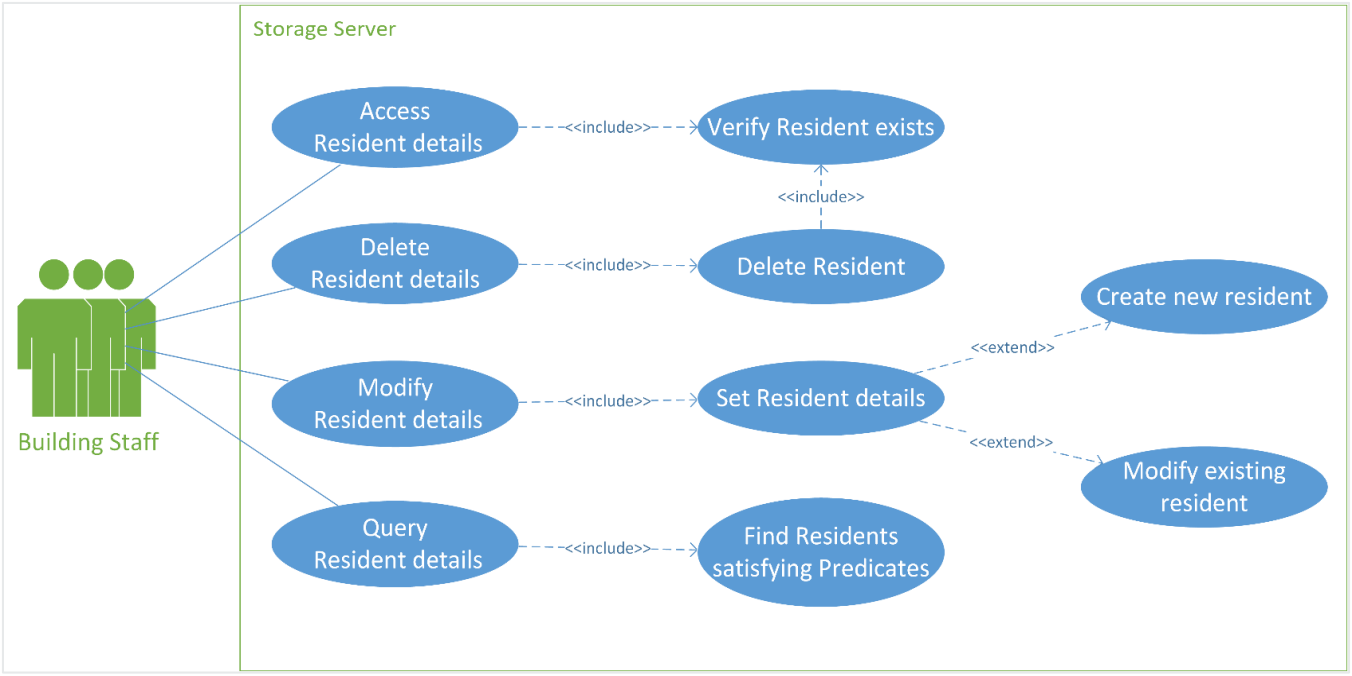

Figure 3: Use Case Diagram - Representation of a user's interaction with the system. Resident details include name, contact details, emergency contact, vehicles, pets, etc.

- **Primary Actor**: Building management office, superintendent, security guards
- **Level**: User level – system only serves 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**: Update tenant information
  1. User connects and attempts to authorize
  2. Server verifies user login credentials
  3. Server requests next command (Get/Set/Query resident records)
  4. User enters desired command (Set)
  5. Server verifies command request and asks for table name, key and value accordingly
  6. Server responds to commands entered after verifying the supplied arguments
- **Extensions**:

5a. User enters non-existent table name	6a. Server notifies user that table does not exist
5b. User enters non-existent key name	6b. A new key/record pair is created
5c. User enters value with incorrect column names	6c. Server notifies user that value entered is invalid

System Requirements

As the developer designs the storage server, he/she must meet certain requirements. These requirements have been developed not only as per the client's desired attributes of the system [1] [3], but also the detailed specifications set by the course instructors [5]. The minimal functionality of the storage server is described by the functions and the major functions that would benefit the client are specified under objectives. The criteria can be used by the designer to evaluate their storage server's functionality against the client requirements. The designed storage server is also expected to respect the limitations imposed under the constraints.

FUNCTIONS

These are the major functionalities required of the storage server to facilitate client-server communication and data storage. The different function calls associated to each function, if applicable, have been mentioned within parentheses.

- Centralize all data to one location and provide multi user connections
- 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)
- 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`)
- 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.

- Ensure server doesn't prematurely respond with success messages
- Manage tenant contact details and facilitate rapid access (i.e. querying (as it requires the most processing) should not be noticeable to the user for less than 10000 records to provide satisfactory performance in user perception)
- Possess an appropriate data structure for multiple record insertion in a hierarchical manner (inserting or modifying storage records in the database 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)
- Reduce amount of paper work that needs to tracked

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)

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 [1] [6]
  - 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 [5]
  - Code must be written in C
  - Client interface file 'storage.h' shall not be changed
  - 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) [7]

Design Decisions

CONCURRENCY

For the last milestone, we have to modify the server such that it can support multiple clients accessing the server simultaneously. Previously, only one client could access the server at a given time, and for another client to access the server, the first client needed to disconnect. Our team aims to modify the server such that it is capable of handling at least 10 simultaneous client connections.

Threads vs. Processes

One of the options to allow for simultaneous connections is to use multiple threads for multiple client connections. A thread is an independent set of commands that run at the same time as the server. Using threads as a concurrency mechanism allows the programmer to divide his/her program into smaller pieces that can be operated and managed independently. Threads allow different processes of the same program to run simultaneously and efficiently on multiple CPUs [8].

In order to implement the new design, the POSIX Threads (Pthreads) library needs to be utilized. The library allows threads to be created and modified; where each of these threads can handle a client connection with the server. This method involves the server acting as shared memory that can be accessed either sequentially or simultaneously by clients [9].

An alternative method for concurrency is to use individual processes for each connection. A process is a set of instructions given by an operating system that is specific to each program code. These instructions fabricate an execution path for each program to follow to ensure proper functionality. Each instruction is part of a thread, thus making threads independent of each other. Every process has its own unique Process ID (PID), code and data (global variables) but could have multiple threads. Each thread has its own unique thread ID, stack counter and program counter [9].

Factor	**Comparison**
Speed	Being part of the same address space in a process allows threads to: 1. Link fast between threads. 2. Take less time to create. 3. Share everything, making an effective communication.
Time	
Communication	
Work	Threads carry out less work as they are used for performing small tasks in a process, whereas process undergo heavy loads of work.

Table 1: Comparison between Threads and Processes [10]

Threads have several advantages over multi-processes as shown in Table 1. However, they could cause problems if commands ‘get’ and ‘set’ are applied to the same index at the same time, possibly resulting

in discrepancies in the database. Therefore, access to shared data is synchronized to make sure that consistency is maintained for all the clients.

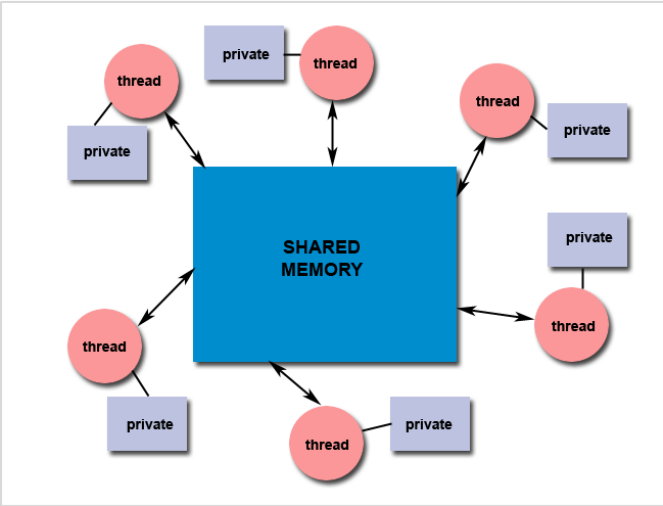

Figure 4: Threads' connection with the Shared Memory [19]

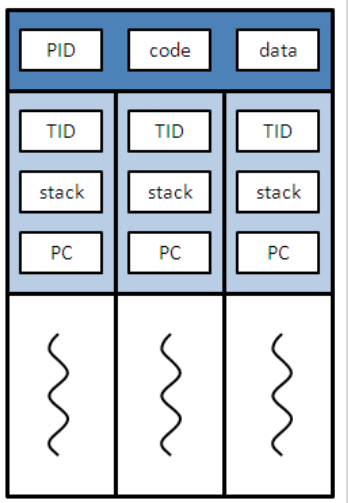

Figure 5: Visualization of Threads [9]

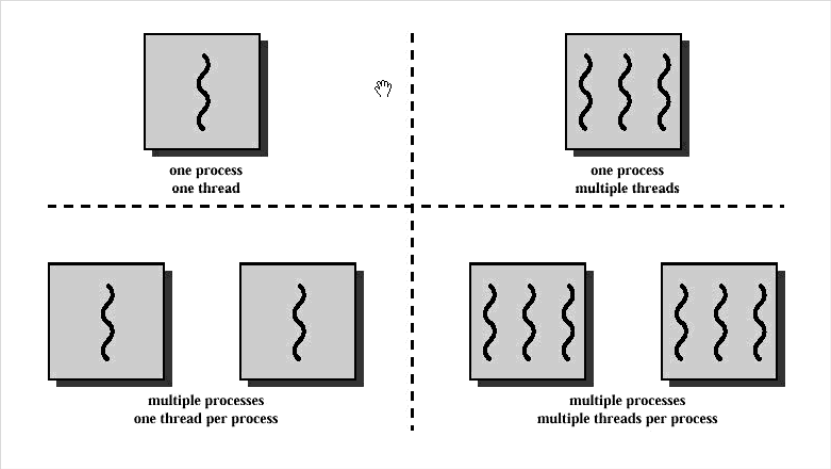

Figure 6: Difference between Threads and Processes [10]

Synchronization – Mutex vs. Semaphore

Synchronization is achieved using tools such as mutex (mutual extension) and semaphore. Mutex is an object that can be locked using a thread at a time. If second thread attempts to lock the same mutex, an error will occur. This prevents different threads from accessing the same mutex. Another way of tackling the overlapping of threads is using semaphore, which is similar to mutex except it involves a counter. When this counter reaches zero, semaphore blocks the access of a thread. The semaphore will keep access to that data blocked until another thread in the process increments the count. There are two parameters used in semaphore: 'uping' (increases the count) and 'downing' (decreases the count) [8].

DESIGN DECISIONS

For the development of this server, our team will use mutex to avoid race conditions [9]. Mutex is secure because it gives sequential access to threads for data; whereas, few threads can access the same data in semaphore. This, in turn, increases the risk of storing incorrect data [11]. On the other hand mutex is easy to implement compared to semaphore as mutex requires only a boolean (true/false) connected variable. When used by first thread to access a data the variable is set to true and when another thread tries accessing the same data, finds connection true, is not allowed to connect. This happens until the first thread returns and makes it false, whereas for semaphore tracking every thread's work on the same data and keeping track of count can be relatively difficult.

Logging

Until now, there were two log files being created: one for client and another one for server. From this point on, two log files will be created per thread to facilitate debugging. Whenever a new client uses the server, separate log files identified by the thread ID will be created.

Implementation

Threads are implemented using "`pthread_create`" function. On every call to this function a new thread is formed with the older thread already running synchronously. Once a function is created, it will pass through "`suspend`" function where the path of thread is decided. If the new thread tries to access the data that is currently being handled by the old thread, the thread will terminate otherwise server allows thread to make fresh changes or access any data in the value of an index. A thread is terminated at the end of "`pthread_create`" function operation or earlier through 'return' in case of any errors [12].

DESIGN DECISIONS

DATA STRUCTURE

The two main qualities of the storage server that concern our client are: speed and the ability to handle large amounts of data [1]. A hash table is a data structure that involves mapping any kind of key to an index that can be used to address elements in the array [13]. A simple example of hash tables is shown in Figure 7. A full comparison of various data structures is presented in Appendix B.

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. The order of time complexity increases with the number of data records being stored [7]. However, this is not the case in hash tables; thus, allowing more data to be stored in the system. Our hash function maps keys to addressable indices; achieving this without any collisions. The implementation of hash tables without any collisions results in optimal performance [14]. We have extended the structure shown in Figure 7 to support residential data by using apartment numbers for keys and data such as tenant name for the entries.

The limitless capacity of hash tables and their performance consistency independent of number of records would best suit our client's needs.

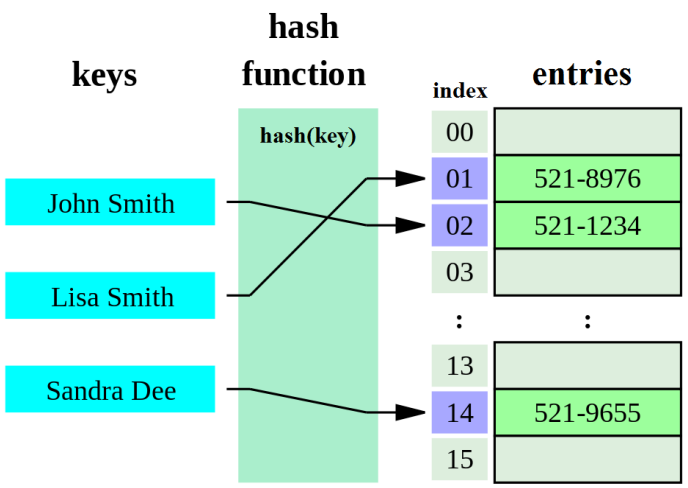

Figure 7: Simple example of a phone book as a hash table. Original image adapted from [13]

PARSING

The general process [15] of how we are parsing the configuration file and all data can be visualized in Figure 8. Samples of how this is applied to a string are shown in Figures 9 and 10.

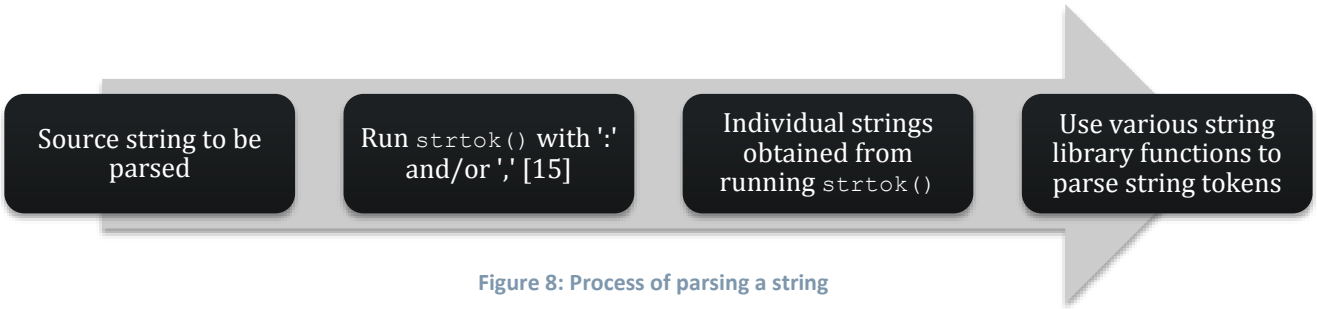

Figure 8: Process of parsing a string

Configuration File

Processing the configuration file involves extracting the connection information and authentication details. The extended configuration file 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]. An example of how a line in the configuration file is parsed is shown below in Figure 9. A sample of the defined format for the server configuration file is given in Appendix A.

DESIGN DECISIONS

Data

Each value at a key in a table is a string containing multiple column-value pairs that are separated with commas [5]. User input is converted into a different format following the defined protocol (Table 2) that our server is capable of decoding. We will be parsing the data manually, without the help of any external tools such as Lex and YACC. This decision was made due to the complexity of implementing Lex and YACC, and the ease of debugging code using C string libraries. An example of how a value inputted by the user is parsed is shown below in Figure 10.

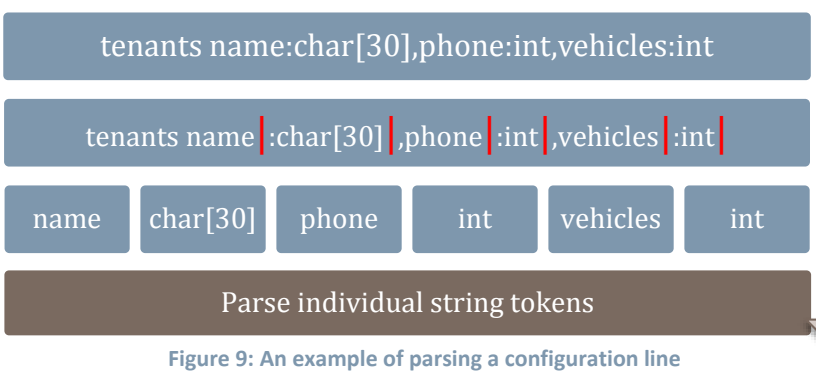

Figure 9: An example of parsing a configuration line

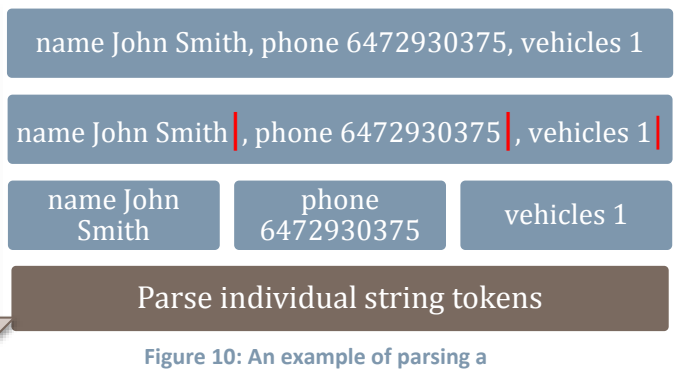

Figure 10: An example of parsing a

CLIENT-SERVER COMMUNICATION PROTOCOL

As communication between the client and server is governed by TCP [16], 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 2 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 #matched_keys #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 2: Client Server Communication Protocol (Rev 3)

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, logged and displayed to the user. Upon experiencing errors due to invalid connection parameters or login credentials, the server disconnects and ends the session. Whereas, errors due to accessing/modifying a record or querying are only flagged and returned to the client: the server does not disconnect. The variety of errors handled will allow the client to report any shortcomings of the system, alongside helping them understand how to use it efficiently. For detailed error codes and their occurrences, refer to Appendix C.

STATE OF THE CONNECTION

To ease interaction between the client and the storage server, once the connection is successfully established with proper authorization, the connection state is reused for multiple get/set requests. It is described using three attributes: connection status, authorization status and a socket file descriptor. If multiple clients with the same server configuration attempt to connect to the server simultaneously, the server will reject the connection.

TESTING

We have ran several test cases on each function in the storage server that have all been successful. Once satisfied with the functionality of individual parts, we tested the entire system for perfect operation. A full list of test cases conducted can be found in Appendix D.

CONCLUSION

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. Following the Agile Development Model, the client's requirements are iteratively assessed against the system's present functionality to best meet the necessary implementation.

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 configuration 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. The project's next step is to improve the performance of the client and server communication, which take's the majority of the end-to-end processing time [17].

Works Cited

[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] ServiceOntario, "Residential Tenancies Act, 2006," 25 March 2014. [Online]. Available: http://www.e-laws.gov.on.ca/html/statutes/english/elaws_statutes_06r17_e.htm. [Accessed 29 March 2014].

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

[8] SoftPixel, "Multithreading in C, POSIX Style," 2006. [Online]. Available: http://softpixel.com/~cwright/programming/threads/threads.c.php. [Accessed 30 March 2014].

[9] ECE297, "Concurrent connections with threads," March 2014. [Online]. Available: https://sites.google.com/a/msrg.utoronto.ca/ece297/tas-and-labs/concurrent-connections-with-threads. [Accessed 30 March 2014].

[10] D. Marshall, "Threads: Basic Theory and Libraries," 5 January 1999. [Online]. Available: http://www.cs.cf.ac.uk/Dave/C/node29.html#SECTION00291100000000000000. [Accessed 30 March 2014].

[11] "Mutex vs. Semaphore, what is the difference?," [Online]. Available: http://koti.mbnet.fi/niclasw/MutexSemaphore.html. [Accessed 30 March 2014].

WORKS CITED

[12] M. Kerrisk, "Linux Programmer's Manual," 18 March 2014. [Online]. Available: http://man7.org/linux/man-pages/man3/pthread_create.3.html. [Accessed 30 March 2014].

[13] Wikipedia, "Hash Table," 20 March 2014. [Online]. Available: http://en.wikipedia.org/wiki/Hash_table. [Accessed 30 March 2014].

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

[15] cplusplus, "strtok," [Online]. Available: http://www.cplusplus.com/reference/cstring/strtok/. [Accessed 30 March 2014].

[16] 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].

[17] CD-037, "Performance Evaluation Report," 17 February 2014. [Online]. [Accessed 29 March 2014].

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

[19] B. Barney, "POSIX Threads Programming," Lawrence Livermore National Laboratory, 11 January 2013. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/. [Accessed 30 March 2014].

Appendices

APPENDIX A – CONFIGURATION FILE

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)
concurrency	An integer that tells the server to turn concurrency on (1) or off (0)

Table 3: Configuration file parameters [5]

Correct Configuration	server_host localhost server_port 1111 username admin password xxxnq.BMCifhU table subwayLines name:char[30],stops:int,kilometres:int concurrency 1
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 concurrency hello
Wrong Configuration	server_host localhost server_port 1111 table subwayLines name:char[30], stops:int, kilometres:int table subwayLines stops:int, kilometres:int
Duplicates of any parameter are unaccepted and will result in an error.	

Table 4: Sample Configuration Files [5]

APPENDICES

APPENDIX B – DATA STRUCTURES

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 5: Data Structures Comparison based on Order of Time Complexity [14]

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 6: Advantages and Disadvantages of different Data Structures [14]

APPENDICES

APPENDIX C – ERROR CODES

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: Meanings of different Error Codes [5]

APPENDICES

APPENDIX D – 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 8: Test Cases conducted to check functionality of Storage Server

APPENDICES

APPENDIX E – 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**
New Design Decisions	**VV**	**VV**	**VV**
Conclusion	**AJ** **PM**	**AJ**	**AJ** **PM** **VV**