Teague Hall (Group 28)
hall1945@umn.edu
CSCI5105 - Introduction to Distributed Systems
Project #2 - Design Document

The project 2 bulletin-board (BB) system was implemented using standard C and POSIX libraries. All networking communications were implemented using TCP/IP sockets with no reference to external libraries.

System Components: There exist three primary system components: **client, server,** and **shared library.**

Client - Provides terminal user interface for listing, viewing, posting, and replying to articles on the bulletin board. User requests are translated into network messages that then route to the BB server(s).

Server - Handles requests from both clients and other servers. Each server hosts its own article database and is responsible for executing the consistency protocol.

Shared Library - All code that is shared between the client and server is placed within a dedicated shared library module. This is particularly useful due to the networking nature of the bulletin-board which requires common knowledge of message formatting.

1. Shared Library: The shared library consists of the following modules that are used by both clients and servers

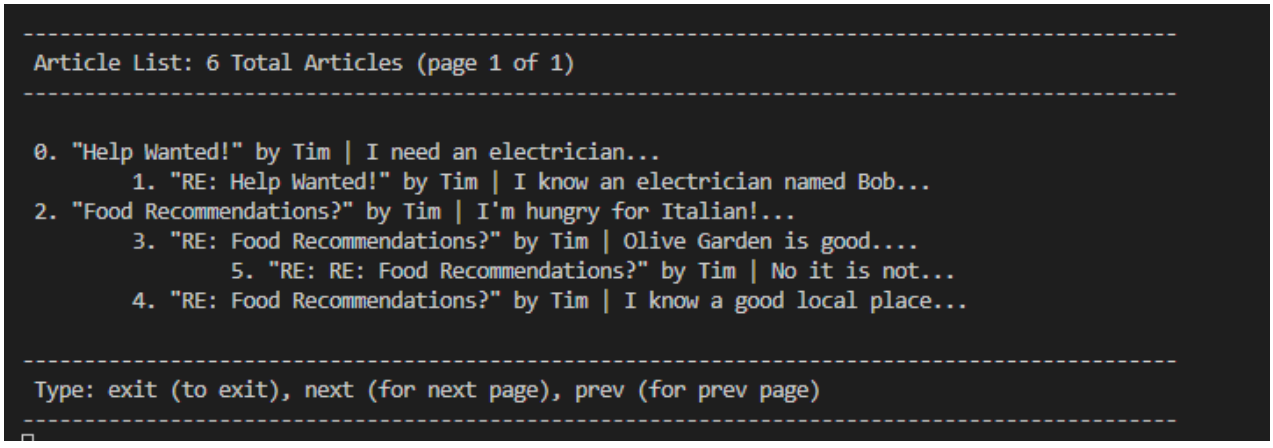
tcp.o : Functions for requesting and accepting TCP/IP socket connections as well as transmitting and receiving data.
msg.o : Functions for building and parsing the application level network protocol.
net.o : Functions for sending the previously built messages across the network
article.h : Defines the common article structure.

2. Client: The client provides the primary user interface to end-users interacting with the bulletin-board.

The client application requires three input arguments:
1. User Name (**required**) - Name that will be attached to all posts and replies.
2. Server Address (**required**) - IP address of server that will be targeted
3. Server Port (**required**) - Port number of server that will be targeted.

Upon successful start-up, users will be prompted with the supported messages and the expected formatting. These messages correspond to the required *Post, Read, Choose,* and *Reply* requirements per the project specification document. The client validates formatting and will respond with correctional actions when appropriate. The client will only connect to the server when a command is submitted. Thus, connection issues to the server will not necessarily show on startup, but rather when the first command is submitted to the server.

The client handles user inputs and provides user-interface displays. An example output from “Read” is featured below.



The client can further display an individual article by submitting the “Choose” command.

3. Server: The server handles requests from both clients and other servers. It is the primary workhouse of the bulletin-board and is thus the most complex component of the system.

The server application requires two input arguments on start-up.

1. Server Address (**required**) - IP address on which server will listen
2. Server Port (**required**) - Port number on which server will listen

After validating the input arguments, the server opens and parses the contents of the local file “config.txt”. This file contains information on the active consistency model used along with the list of the servers that comprise the server group. If the inputted listening server arguments match the coordinator server specified in the configuration file, that server is able to identify itself as the coordinator. Please refer to READE.md for more information on configuration specifics.

3.1. Server Components The server consists of the following modules

server.o : The main entrance point to the server. This function validates input arguments, parses the configuration file, and starts an infinite loop accepting incoming connections. Upon each connection accept, a new thread is spawned for the connection handler (see below).

connection_handler.o: Contains the handler routine for each newly accepted connection. This routine reads the remainder of the message and forwards it to the corresponding message handler (see below).

msg_handler.o: Contains the routines to respond to each message the server supports (e.g. post, reply, etc.). This module will either directly handle message requests (e.g. update local database per the received replica) OR route the request to another function depending on the active consistency protocol.

protocol_sequential.o: Contains routines associated with the sequential consistency model.
protocol_quorum.o: Contains routines associated with the quorum consistency model.
protocol_readyourwrite.o: Contains routines associated with the read-your-write consistency model.

3.2. Consistency Models: The server supports three consistency models which are detailed below:

Sequential Consistency - Sequential consistency was implemented via the primary-backup protocol using a non-blocking replication acknowledgement. In the case where a client contacts a non-primary server, the following sequence occurs.

1. Client contacts non-primary server with post request.
2. Non-primary server forwards post request to coordinator.
3. Coordinator receives post request and updates its database replica.
4. Coordinator responds to the non-primary server with acknowledgement // BEFORE REPLICATING TO OTHERS
5. Non-primary server then responds to client that operation was successful
6. Coordinator sends replicas to other servers

Read-Your-Write Consistency - Read-Your-Write consistency was implemented again using the primary-backup protocol but this time replicating the updates to all other servers prior to responding to the client request.

1. Client contacts non-primary server with post request.
2. Non-primary server forwards post request to coordinator.
3. Coordinator receives post request and updates its database replica.
4. Coordinator sends updated database to each non-primary server and waits for acknowledgement
5. Coordinator then responds to the original non-primary server with acknowledgement.
6. Original non-primary server then responds to client that operation was successful

Quorum Protocol Consistency - Quorum consistency is implemented using the previously gathered Nr and Nw values from config.txt along with the general message functions. The sequence is detailed below.

1. Client makes a read or write request to the server.
2. For reads, the server will reach out to Nr servers and grab the highest version data replica received.
3. For writes, the server will forward the write to Nw servers.

4. After successfully reading Nr or writing Nw, the server will respond to the client to indicate success.

When operating under the quorum protocol, a background thread checks neighboring replicas for newer versions. If a newer replica version is found, that new version will be synchronized.

3.3. <u>Consistency Analysis</u>: Theoretical timing analysis of the three consistency models and their comparison are provided below. (programmatic studies were originally included as part of this project but were later abandoned due to limited time)

<u>Read-Your-Write via Blocking Primary-Backup Protocol</u>

$Client\ Read\ Time\ =\ T_{CS}$
$Client\ Write\ Time\ =\ T_{CS} + (N - 1)\ *\ T_{SS}$ where *N* is the number of nodes, T_{CS} is the latency between client and the primary server, and T_{SS} is the latency and computation time for server to server replication.

<u>Sequential Consistency via Non-Blocking Primary-Backup Protocol</u>

$Client\ Read\ Time\ =\ T_{CS}$
$Client\ Write\ Time\ =\ T_{CS}$ where only the client to primary server latency and computation time are featured. The primary server acknowledges the receipt primary to replicating the update to all other databases. Thus, the client response time has improved versus the blocking implementation, however, it is possible that the client will not see its previous write when connecting to a new server.

<u>Quorum Consistency Protocol</u>

$Client\ Read\ Time\ =\ T_{CS} + (Nr)\ *\ T_{SS}$
$Client\ Write\ Time\ =\ T_{CS} + (Nw)\ *\ T_{SS}$

The quorum consistency protocol is of particular interest when considering the read times. A single data read now requires checking Nr servers when we previously were only checking one. Writes are also a bit more expensive but have improved fault tolerance since we only need to write to Nw (< N) nodes.

4. <u>Testing</u>: Testing was performed to validate the operation of the client, networking, and consistency models under normal and abnormal operating conditions. This test was done manually (i.e. not programmatically) described below:

<u>Client - Normal Operating Conditions</u>: Validated that client user-interface responded correctly to user inputs, properly displayed articles (per indentation requirements), and correctly submitted messages to a single server.

<u>Client - Abnormal Operating Conditions</u>: Validated that improper user inputs were caught and communicated back to the user. Network issues (e.g. incorrect or downed server) are also identified and communicated to the user.

<u>Consistency - Normal Operating Conditions</u>: For each consistency model, validated that the POST, READ, CHOOSE, and REPLY successfully replicated to all servers. For the coordinator protocols, replication was validated by writing to both the primary and non-primary instances and ensuring no errors.

<u>Consistency - Abnormal Operating Conditions</u>: Validated that each consistency model responded appropriately to network interruptions or downed nodes. Specifically, for the primary-backup protocols, errors were communicated back to clients in the event that a server failed. Similarly for the quorum protocol but in the event that multiple nodes had failed.

5. <u>Pledge</u>: I, Teague Hall, pledge that this project along with the source code is original work. No code was copied from external resources.