P2 - Ring Topology of servers

May 1, 2022

Contents

1 Code Details

1.1 Broad View

Model is stateless but TCP is used because we don't want any loss of data. If loss of data can be tolrated, model can be easily shifted to UDP with minimal change to code.

1.1.1 Compilation

makefile: Used to compile the code. There are three options:

1. compile: To compile the code. This is default option for make.
2. info: Used to display the status of ports of the system.
3. clean: Used to clean all files that are generated by makefile.

makefile will create the following folders

1. bin folder: which will contain the executable code for Broker, Subscriber and Publisher.
2. logs folder: Every event is logged here.

1.1.2 Execution

To run, go to bin folder.

1. For broker: ./Broker.o <N = number of servers> <start port = starting port number for servers to initlialise>
2. For subscriber: ./Subscriber.o <IPaddress = server> <PORT = server>
3. Similar for producer

1.1.3 Logging

Every subscriber, producer and broker will generate a log file of all the events that are taking place. Log also includes the sequence of messages transferred in network. Each broker is identified by the port number it is operating (as all of them are on same system) and log file will be named as <port_number>.log

1.2 Constants

1. BULK_LIMIT: inside helpers/Database.h
2. MESSAGE_TIME_LIMIT: inside helpers/Database.h. Represents time in seconds.

1.3 Helpers

1. helpers/Database

   This is the databasae which is used by each server. The database provides the API for common operations that are required to be performed.

   Database is implemented with singleton pattern, so it is available across files in server. It has following members:

   (a) void removeOldMessages(const string &topic): This is called to remove those messages which are expired andd are related to topic *topic*.

   (b) static Database& getInstance(): Called to get the instance of database (the class is singleton)

   (c) int addTopic(const char* in_topic): Add the topic *in_topic* to the database. returns -1 in case topic already exists in database else returns 0 meaning success.

   (d) bool topicExists(const char* in_topic) const: Checks whether topic *in_topic* exists in database or not.

(e) addMessage(const char* in_topic, const char* in_message, short_time& time): For topic *in_topic* check if exists. If not, returns -1. Remove old messages related to the topic and insert the message with time stamp given by *time*. Returns 0 if topic already exists and it denotes success.

(f) addMessages(const char* in_topic, const vector<string> &msgs, short_time& time): Do the same work as addMessage but in bulk. return value have same meaning.

(g) getNextMessage(const char* in_topic, short_time &clk): If topic *in_topic* doesn't exist or no more messages are there which were inserted after *clk,* empty string is returned. Else the database is searched for message with time more than or equal to *clk* and returned.

(h) getBulkMessages(const char* in_topic, short_time &clk): does the same job as getNextMessage but returns array of { time, message } where *time* is the time when *message* was inserted into database for topic *in_topic.* Number of messages returned is limited by *BULK_LIMIT.*

(i) getAllTopics(): return the array of all topics that are registered in database.

2. helpers/Queue

   Use system message queues underneath and do all dirty work of opening and closing queues in system.

3. helpers/Time

   (a) current_time(): returns current time in C++ time format.

   (b) currentDateTime(): returns string representation of current date and time.

   (c) DateTime(short_time time): returns string representation fo *time*.

4. helpers/ThreadPool

   As the name implies, it manages the execution of multiple threads and assign them tasks. A new operation is scheduled by *startOperation(connfd, lambda*) where connfd is the connection fd which is returned to first parameter of lambda. Second parameter of lambda is threadId which is used to get mutex and condition variable associated with executing thread.

5. helpers/SocketLayer

   (a) MAX_CONNECTION_COUNT represents maximum number of connections that server supports from outsid world: subscribers and publisher combined.

   (b) maxMessageSize: Maximum length of message that can be sent.

   (c) maxTopicSize: Maximum length of topic that can be prescribed. The value is chosen as 26 based on the padding of struct that are defined later for communications.

   (d) SocketInfo: Struct which contains the information for each connection.

      i. sockfd: fd of socket that is used for communication.

      ii. connfd: connection fd.

      iii. my_addr: my local address

      iv. dest_addr: destination address

   (e) MessageType: Possible types of messages that can be there while communication between brokers or broker and Publisher or broker and Subscriber.

   (f) MSGHeader: Header of any message that is used for communication.

      i. isLast: When the current message is last of consecutive messages, it is set to true. Usage explained later.

      ii. payload_size: Size of current payload that we need to read from socket for client/server communication. Required because of variable message length.

   (g) ClientPayload: The message that is used for communication b/w Broker and Pulisher/Subscriber.

      i. time: Meaning varies based on communication. Explained later.

      ii. msgType: value of *MessageType* enum.

      iii. topic: string that represents the topic of communication.

      iv. msg: Message of communication.

(h) ServerPayload: The message that is used for communication b/w brokers in ring topology.

i. sender_server_port: Used to uniquely identify the server which generated the request. Should be replaced by sockaddr if brokers are sitting on separate systems.

ii. sender_thread_id: When the message is received and it is generated from current system, it is sent to the thread with thread id *sender_thread_id* which is the thread who sent the request to neighboring servers.

iii. client_payload: copy of the client request that was sent to original server along with replies of servers, explained later in detail.

(i) PresentationLayer: Attaches MSGHeader and formats data so that Broker/Subscribers/Publishers code need not to worry about communication format.

i. getData(const int connfd, string &errMsg, bool &connectionClosed): actively listen for packets on the connection fd *connfd.* Sets errMsg in case of error to report and sets connectionClosed in case the connection is closed. returns array of ClientPayload -> Not used for Broker - Broker communications. If error, array returned will have 0 size.

ii. sendData: reverse of getData. returns -1 in case of any error and errMsg is set appropriately for logging.

iii. getServerReq: same as getData but for Broker-Broker Communication.

iv. sendServerData: same as sendData but for Broker-Broker Communication.

1.4 Subscriber.cpp

Execution sequence (main)

1. Create log file and redirect stderr to logs/Subscriber.log
2. Get input parameters.
3. Create connection with server.
4. Register itself as subscriber to the server by sending a message to server – So that server can distinguish the connection type and branch into appropriate function (explained later)
5. If response of server is OK, call do_task otherwise exit with error message on terminal.

Execution sequence (do_task)

- Initialise the subscriber object and perform tasks.

Subscriber Class

- Communicates with the server and read presentation data. Function names are self explanatory in code.
- *topic:* preserved across requests.
- *t*: Initialised to current time - 60 seconds (time limit). It is used to keep track of how many messages are requested till now. For each reply from server, this is updated.

1.5 Publisher.cpp

Have similar code as Subscriber.cpp

1.6 Broker.cpp

1.6.1 Struct ServerInfo

1. q: Message queue, used for synchronization during initialisation
2. thread_pool: self explanatory
3. send_mutex: mutex for server main thread

4. PORT, server_send_port, server_recv_port: initialised in main after fork. Meaning explained there in doc.
5. sendServerPORTInfo: socket info to send data (Broker-Broker communication)
6. listenSockInfo: sokcet info for receiving data (Broker-Broker communication)

1.6.2 Initialisation (Parent)

1. Read command line args and initialize N servers. Each server is initialised to have:
   (a) Message queue, that is used for synchronization during initailsation.
   (b) Current port number to use.
   (c) Port number of neighbor on which to send data.
   (d) Port number of neighbor from which we need to receive data.
2. Wait for all servers to spawn.
3. Send signal to servers to start making connections with each other using msg queues.
4. Wait till connections are established. After that, wait for all servers to exit and then return.

1.6.3 Initialisation (Spawned Servers)

1. On call to serverOnLoad from parent process, SIGCHILD is registered
2. create listenSocketInfo as passive socket so that it can be connected later with brokers.
3. Establish connection with neighbors after synchronization with parent.
4. Initialise thread pool and start a thread to listen from other Broker.
5. Create and redirect stderr to log.
6. Wait for connections from clients. For each client, read the type of client (sent from client initially) to assign tasks.

Other function names are self explanantory and their overall details are explained in next section.

2 Data Flow

2.1 Publisher - Broker Communication

Request	Reply	MSGHeader		ClientPayload			
		isLast	payload_size	time	msgType	topic	msg
Create Topic		1	48	-	CREATE_TOPIC	Topic	-
	Invalid topic name	1	48	-	INVALID_TOPIC_NAME	sent Topic	-
	Topic already exists across all servers	1	48	-	TOPIC_ALREADY_EXISTS	sent Topic	-
	OK	1	48	-	SUCCESS	sent Topic	-
Push message		1	calculated	current time	PUSH_MESSAGE	Topic	message
	Topic doesn't exist across servers	1	48	time sent from client	TOPIC_NOT_FOUND	sent Topic	-
	OK	1	48	time sent from client	SUCCESS	sent Topic	-
Push file		0000..1	calculated	unique to all	PUSH_FILE_CONTENTS	Topic	messages
	Topic doesn't exist across servers	1	48	time sent from client	TOPIC_NOT_FOUND	sent Topic	-
	OK	1	48	time sent from client	SUCCESS	sent Topic	-

2.2 Subscriber - Broker Communication

Request	Reply	MSGHeader		ClientPayload			
		isLast	payload_size	time	msgType	topic	msg
Get all topics		1	48	-	GET_ALL_TOPICS	-	-
	No topic is registerd across servers	1	48	-	TOPIC_NOT_FOUND	-	-
	OK	0000..1	48	-	SUCCESS	topics per reply	-
Get next message		1	48	t (see reply)	GET_NEXT_MESSAGE	Topic	-
	No more messages	1	48	no message with time > t	MESSAGE_NOT_FOUND	sent Topic	-
	No topic is registerd across servers	1	48	time sent from client	TOPIC_NOT_FOUND	sent Topic	
	OK	1	calculated	time when msg was entered	SUCCESS	sent Topic	msg
Get Bulk Messages		1	48	t (see reply)	GET_BULK_MESSAGES	Topic	-
	No more messages	1	48	no message with time > t	MESSAGE_NOT_FOUND	sent Topic	-
	No topic is registerd across servers	1	48	time sent from client	TOPIC_NOT_FOUND	sent Topic	
	OK	0000..1	calculated	time when that msg was entered	SUCCESS	sent Topic	msgs

2.3 Broker-Broker Communication

For broker to broker, table entries are trivial.