Project Title: A Pull-based Cache invalidation

Instructions on how to run the program:

1. Export the c file proj_inval.c to willow server. This can be done using winSCP.
2. Compile the file using the command, "csim64.gcc proj_inval.c –o proj_inval".
3. Run the executable file using the command, "./proj_inval".
4. The program must be run several times, for the first graph Cache hit vs. T_update, set T_query = 10 and T_update = 5, 10, 20, 100, 200 in 5 runs respectively.
5. For the second graph Query delay vs. T_query, set T_update = 10 and T_query = 5, 10, 20, 100, 200 in 5 runs respectively.
6. For the third graph Query delay vs. T_update, set T_query = 10 and T_update = 5, 10, 20, 100, 200 in 5 runs respectively.
7. For each run, the results are plotted in a graph which is analyzed below.

Functionalities:

create_db() and **create_cache(i)** are used to create the database (DB) and 5 caches for 5 clients and initialize them.

data_update() updates the last_update_time of the data items based on mean update arrival time and hot data update probability.

server() creates the server process and waits for a TIME_OUT period of 20s to receive any message in its mailbox. If an event occurs i.e., a message is received into its mailbox from a client, then it checks the type of message. If the message type is REQUEST, then it forms a DATA message and sends it to the corresponding client. If the message type is CHECK, then it compares the data item's last_update_time included in the CHECK message with the data item's last_update_time in the DB. If DB's last_update_time is more than the CHECK message's last_update_time, this implies that the data item is updated in the DB. So, server sends a DATA message including the new last_update_time to the client. If both are equal, then the data item cached in the client is valid so it sends an ACK message and continues to wait for new messages from the clients.

init() initializes all the facilities, mailboxes needed for the project. Once the simulation time ends, the values of queries generated and cache hits are computed and displayed.

client(i) creates the 5 client processes and waits for a TIME_OUT period of 20s to receive any message in its mailbox. During TIME_OUT, it holds for T_query and then randomly picks a data item based on the mean query generate time and hot data access probability. It generates a query for the data item. Then it checks its cache using check_local_cache(). If the data item is cached, then it sends a CHECK message to the server including the data item's last_update_time cached. If the data item is not cached, it sends a REQUEST message to the server. In both the cases, the number of queries is incremented by 1. When an event occurs, i.e., a client receives a reply from the server, it checks the type of the message. If the message type is ACK, then the cached copy is valid and cache hit is incremented by 1. Else the message type is DATA. Here, the client checks if the data item is previously cached or not. If it was already cached, it simply updates the las_update_time value in the cache. Else, it checks the cache status whether it is full

or not using caching(). If the cache is not full, then the new data item is simply added into the cache. If it is full then cache_replacement() is called.

check_local_cache() checks if the data item is cached and returns the index at which it is cached. Else it returns -1.

caching() increments over all the elements and when there is an empty slot, it stores the data in it. If the cache if full, it returns -1. Once the cache is full, the number of queries generated and cache hit values are set to zero and incremented again. This is done to remove the cold state.

cache_replacement() is called when the cache is full. It compares the last_access_time of all the data items in the cache and replaces the data item with the oldest last_access_time following the LRU policy.

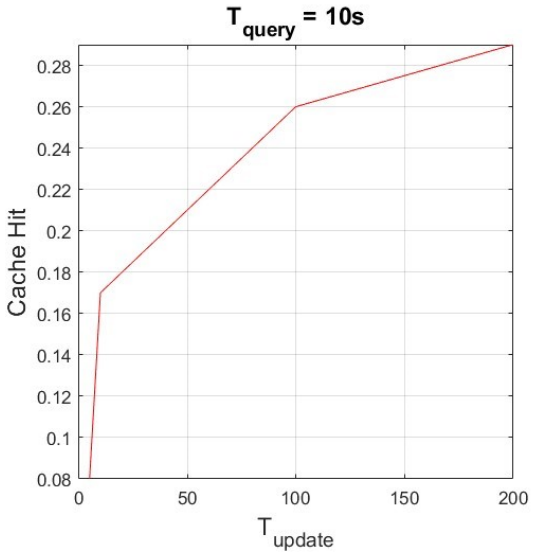
cache_update() updates the last_updatet_time of an already cached data item, if it is updated in the DB.

new_msg() creates a new message which is sent by the client to the server.

send_msg() sends the message from source to destination included in the message. It holds for 0.8192 + 0.0079 when the message is DATA message and 0.0079 in case of other messages. This is because data item has a size of 8192 bytes and the rest of the message has a size of 79 bytes (it is shown as comments in the code). **form_data_msg()** appends data into the original message. **form_ack_msg()** is an acknowledgement.

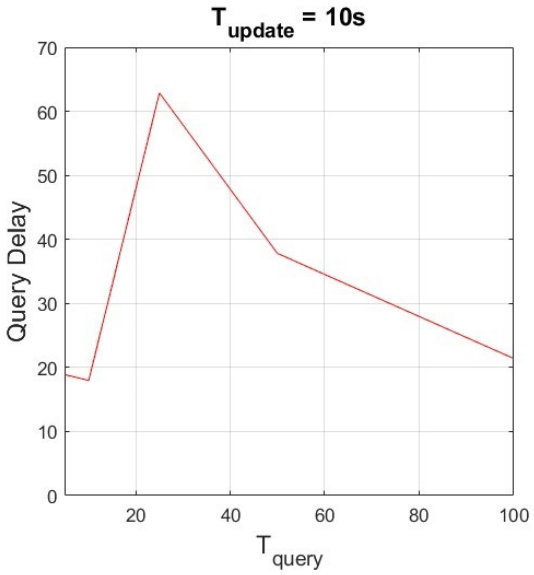
Results:

Graph 1: T_update vs Cache Hit where T_query = 10s (constant)



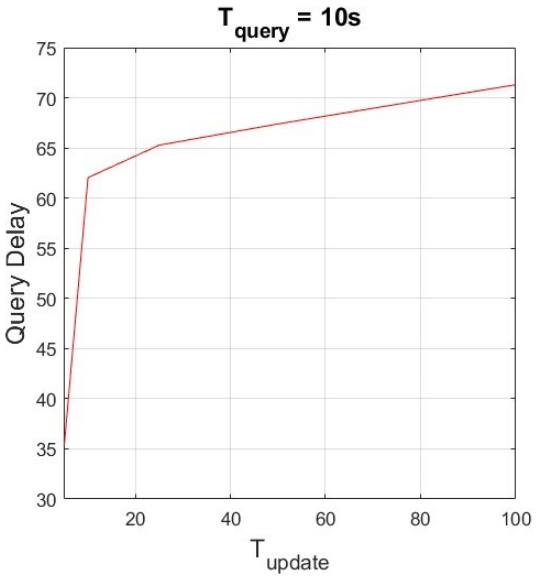
As the mean update arrival time increases in the order of 5, 10, 20, 100, 200s, the amount of time it takes for the data update increases. So the data items cached in the clients are valid for longer time. Hence the cache hit ratio increases as seen in the graph.

Graph 2: T_query vs Query Delay where T_update = 10s (constant)



As the mean query generate time increases from 5, 10, 25, 50, 100s, the number of queries generated become less. So, the server has to respond to fewer messages. As a result, number of messages in the queue for the server to respond become less. So the query delay reduces. The sudden increase at T_query = 25s needs to be analyzed further.

Graph 3: T_update vs Query Delay where T_query = 10s (constant)



As the mean update arrival time increases from 5, 10, 20, 100, 200s, the update frequency decreases. So cache hit increases and Query delay decreases but I'm getting opposite results here.

Assumption and implementation issues:

Once the cache is full, the number of queries generated by a client and the number of cache hits are set to zero and incremented again from zero. This is done to remove the cold state.

I have executed the code for a simulation time of 2000. For T_update = 5, 100, 200 the code is failing when the simulation time is set as 50000. It is running perfectly for T_update = 10, 20. So, to keep the graph values constant, I ran the simulation for only 2000s.