CIS 415 Operating Systems

Project 3 - InstaQuack!

Submitted to:

Prof. Allen Malony

Author:

Michael O'Connell

Report

Introduction

InstaQuack! is a social media service that provides users a way to communicate what they are doing using real time photos with short captions. The project uses the publish/subscribe model (Fig 1) to allow data (photos) sharing between content creators (publishers) and subscribers. Publishers send their data to a broker which stores it under a given (specified by publisher) topic. Subscribers can then receive the data from the broker for the topics they are subscribed to.

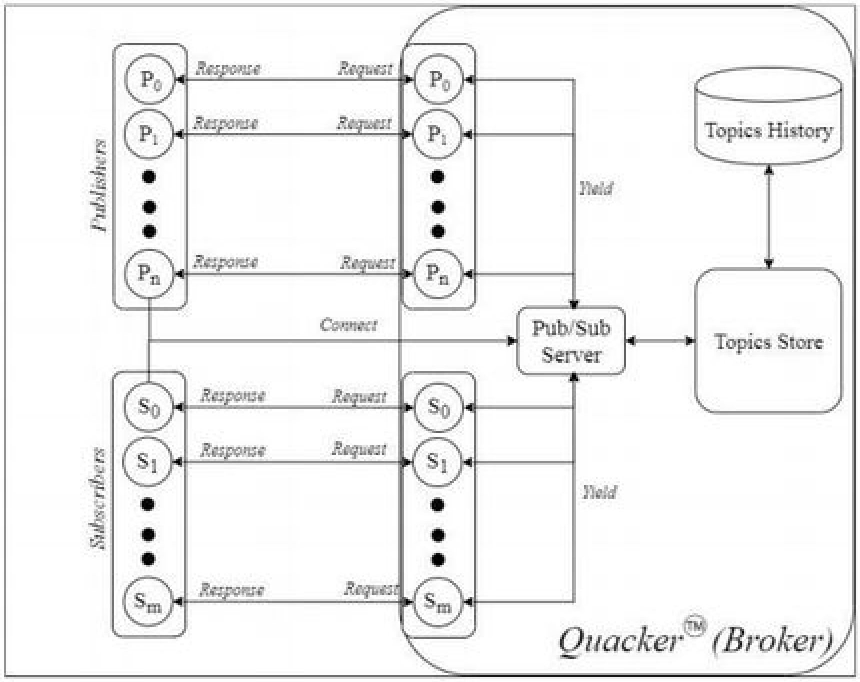

Fig. 1: InstaQuack! Architectural Diagram

The *Quacker Topic Store* is where recently published photos are stored. The Quacker server serves as an intermediary for publishers and subscribers to store/receive photos. The server creates threads to perform work for publishers and subscribers. The interface facilitating client-server interaction is the series of command files for a given publisher or subscriber (client). The server gives these files to its appropriate threads (publisher or subscriber) and they interpret the files and perform the commands. Before this can happen, the server also has to be initialized using a main command file (which is interpreted and the initialization commands are performed). Finally, once the server has been initialized and the publisher/subscriber threads have been created, subscriber threads use the gathered data to create the *InstaQuack Topic Web Pages* (dynamic html pages). This allows a client to open each of the files and see what each subscriber was able to get from the topics.

Background

The *Quacker Topic Store* consists of a global array of "topic queues" (held in "registry") holding "topic entries" and has associated methods, "enqueue()", "dequeue()", and "getEntry()", for usage of each queue. More specifically, each queue is a circular ring buffer that holds "topic entry" structures. The associated methods are as follows: enqueue() adds or "pushes" entries on to the queue, dequeue() removes or "pops" entries from the queue, and getEntry() simply gets entry's content or "peeks" entries from the queue.

The Quacker server consists of a thread pool for publishers (held in "publisher_thread_pool") and a thread pool for subscribers (held in "subscriber_thread_pool"). A maximum number of threads are available (held in "NUM_PROXIES", where publisher and subscriber each a maximum of half of this value) to be created for publisher and subscribers (as dictated by the program's command file). If there are more publisher/subscriber requests then there are available threads, the requests are held in a "job overflow queue" (a circular ring buffer with slightly different members than topic_queue), which has its own data structure and methods. Once a publisher/subscriber finishes it's current request, it can pull from the overflow queue to perform the next request. This pattern continues until there are no more requests.

The server initialization comes from a main command file through standard input. The main section parses the command file and performs the commands (creating threads, setting the delta amount. etc). The threads are created and have their associated command files passed in through a thread structure. The threads themselves then parse their command files and perform the commands (enqueuing, get_entrying, etc.).

Subscribers create and modify an html file using the data collected from their "get" (get_entry()) commands. If a subscriber picks up a new job, it creates a new html file to represent its new findings.

Implementation

The *Quacker Topic Store*'s queue has members "name" and "topic_id" for identification, "buffer" is the actual "topic entry" queue (circular ring buffer), "head" and "tail" are for keeping track of the beginning and ending of the queue, "max_entry" and "entry_cnt" are for keeping track of how full the queue is, "overall_entry_cnt" is used for assigning unique and monotonically increasing "entry_num" identification for each "topic entry" in the "buffer" queue, and "lock" is a mutex for the queue ensuring exclusive access. Each "topic entry" in the queue has members "entry_num" for identification, "time_stamp" for age of the entry, "pub_id" for the thread identification regarding "who" pushed the entry to the queue, and "photo_url" and "photo_caption" for description. enqueue() takes in a topic_id and an entry, it pushes entry into the queue, it uses the "topic_id" to find the correct registry index for the desired topic queue (implemented in a helper method, get_registry_index()), checks whether the queue is full, updates the entry's timestamp and "entryNum", pushes the entry at the "head" position, and updates head. It uses the topic queue's "lock" member to ensure exclusive access to the queue. get_entry() takes in a topic_id, a last_entry, and an entry, it "gets" an entry from the queue that has an entry_num greater than or equal to last_entry + 1 by copying the content of the entry found to the empty entry passed into the method (by reference). get_entry() finds the correct registry index for the topic queue with a matching topic_id (using get_registry_index()), checks whether the queue is empty, and then goes through the queue starting a the oldest entry (the queue's tail) and ending at the newest entry (the queue's head) looking for last_entry + 1. If it finds an entry with a entry_num greater than last_entry + 1, it copies that entry's contents and returns that entry's entry_num (this occurs because last_entry + 1 was dequeued by our cleanup thread). If it finds an entry with a entry_num of last_entry + 1, it copies that entry and returns 1. If the topic queue is empty or it doesn't find an entry_num equal to or greater than last_entry + 1 (meaning it hasn't been entered yet), it returns 0. It uses the topic queue's "lock" member to ensure exclusive access to the queue. Lastly, dequeue() takes no arguments and goes through all entries in all topic queues in the registry and removes the entries with an age greater than delta (a global variable in seconds set by the command input). dequeue() uses a helper method, compare_age_and_pop(), to calculate the age of an entry, compares it to delta, and increments the tail (effectively popping the entry) if the age is greater. It uses each topic queue's "lock" member for exclusive access to the queue.

The Quacker server is dependent on the program command file, in the sense that threads are assigned to jobs every time an "add" command is read. If more jobs are requested than available threads, the outstanding jobs are put in the overflow queue (see **background**). If less jobs are requested, the thread amount created will be equal to the number of jobs. The creation of threads and pushing outstanding jobs to the overflow queue occurs before the "start" command is read in the command file. Once a publisher/subscriber thread is created, it is halted (using pthread_cond_wait()). Once all threads are created and halted (main uses sleep() for 5 seconds to ensure this actually occurs), they are started at the same time (resuming them is implemented using pthread_cond_broadcast()) so they run concurrently. All threads are finished running when they've finished their assigned job (implemented using a flag set when end of file has been reached), no jobs remain in the respective overflow queue (implemented using the return value of dequeuing the overflow queue) , and the end of the command file has been reached (i.e. no more jobs can be sent to the overflow queue; this is implemented using a global flag). Ensuring all threads have finished running is implemented using pthread_join(). Lastly, a cleanup thread is also created and ran alongside the publisher and subscriber threads. It periodically (every delta seconds) goes through each of the topic queues in the global registry and removes all entries greater than delta. It makes the comparison between entry age and delta using gettimeofday().

The parsing of the main command file and the publisher/subscriber command files is accomplished using str_filler() in string_parser.c, converting string values found in the files to usable integer values is accomplished using atoi() and sprintf().

Outputting to the associated html file in subscriber is accomplished using fprintf(). Naming the file is accomplished using strcat() and sprintf().

Performance Results and Discussion

I'll be referring to screenshots in the project3.tar.gz file because there are too many screenshots to reasonably post in this document.

Program compiles successfully (see screenshots folder).

Topic queue and associated method behavior is successful:

- enqueue() successfully gives notice if topic queue is full (see tests/pub_dont_give_up_test) and updates head and entry_cnt successfully (see tests/normal_test and look at the head/entry_cnt values for a successful enqueue()).
- get_entry() successfully implements case 1 - topic queue is empty (see tests/sub_empty_test). Successfully implements case 2 - last_entry + 1 is in the topic queue (see tests/normal_test). Successfully implements case 3 - last_entry + 1 has yet to be put in the queue and topic_entry + 1 was deleted by dequeue (see tests/sub_empty_test).
- dequeue() successfully removes entries greater than or equal to delta in a periodic way (see tests/normal_test).

Program reads the main command input successfully and performs the commands successfully (see tests/normal_test).

Quacker server threads are allocated successfully and exit appropriately (not prematurely) (see tests/normal_test). Threads also read command input successfully and perform the commands successfully (see tests/normal_test).

HTML output files are created and formatted successfully (see .html files in normal_test).

Concurrency among threads is demonstrated through the variability of HTML file output (see .html files in normal_test AND .html files in normal_test/alternate_html).

Conclusion

Through this project I learned a tremendous amount about mutual exclusion and concurrency and gained an appreciation for multithreading as a technique to increase concurrency.