Project #2
Quacker: A Better Twitter for Ducks Only!

CIS 415 - Operating Systems
Fall 2015 - Prof. Malony

Due date: November 30, 2015, midnight

Introduction

Today, social networking is where it is at. Puddles, the Oregon Duck and a social animal himself, was browsing the University of Oregon's **Around*the*O** webpage and got a thought about how to take it to the next level. Puddles's idea was to create a social networking platform that would appeal to the UO undergraduates that make up his fan base. After consulting his "branding" manager, Puddles is calling his new social media service **Quacker**! The goal is to have a way for the UO student community to communicate news of vital importance, as it is happening LIVE. A cross between Twitter and SnapChat, Quacker will have proprietary server technology called *QuackIt* that can connect news producers (like Puddles) with news consumers (like his Fan Club), but with the twist that new news will always have faster access than old news. Now, all he needs is to recruit OS developers from Prof. Malony's CIS 415 class to make it happen.

At the heart of many systems that share information between users is the *publish/subscribe* model. The idea is that *publishers* of information on different *topics* want to share that information (articles, pictures, ...) with *subscribers* to those topics. The publish/susbscribe (*pub/sub*) model makes this possible by allowing publishers and subscribers to be created and operate in a simple manner: publishers send the pub/sub system information for certain topics and subscribers receive information from the pub/sub system for those topics that they are subscribed to. There can be multiple publishers and subscribers and they all may be interested in different topics.

Quacker must be responsive, scalable, and rival anything that can be found at other Pac-12 schools. With the extraordinary skillset that you are learning in CIS 415, you will implement the heart of Quacker – the QuackIt pub/sub server architecture – shown in Figure 1).

There are 6 parts to the project, each building on the other. The objective is to get experience with a combination of OS techniques in your solution: interprocess communication, threading, synchronization, and file I/O. Althought the parts are described in the order below, you should start with Part 4. It will be discussed in detail in the lab. A skeleton for Part 1 will be provided to you.

Part 1 – Registering with QuackIt

Part 1 is to develop the first version of the QuackIt server that makes it possible for a publisher or subscriber to connect to the service. The idea is that a publisher or subscriber first needs to contact QuackIt to register themselves and indicate what topics they are interested in. This will be done through interprocess communication (IPC) using pipes. To keep things simple to begin with, Part 1 implements the following:

- Write a main program that forks the QuackIt server process, n publisher processes ($P_i,\ 1 \leq i \leq n$), and m subscriber processes ($S_j,\ 1 \leq j \leq n$). n and m are command line arguments.

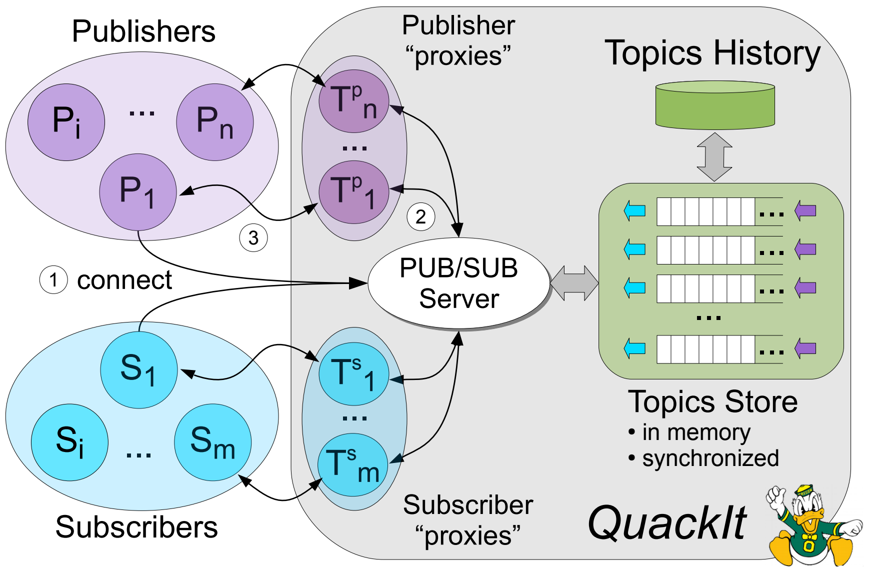

Figure 1: Quacker and it QuackIt pub/sub server architecture.

- The QuackIt server creates a pipe with each publisher and each subscriber process. It then listens for information on each pipe.

- When each publisher, P_i, starts up, it will `exec()` a program that will be provided to you. Each producer then uses the pipe it inherited to communicate with the QuackIt server and send messages. Once the new publiser program is running, it sends the following messges:

  "pub pubid connect"
  "end"

  After each message string, the publisher waits for a response from the QuackIt server before sending the next string. If the response is "accept" then the next string can be sent. If the response is "reject" then something went wrong and the publisher should try again or terminate.

- Similarly, when each subscriber, S_j, starts up, it will `exec()` a program that will be provided to you. Each subscriber then uses the pipe it inherited to communicate with the QuackIt server and send messages. Once the new subscriber program is running, it sends the following messges:

  "sub subid connect"
  "end"

  After each message string, the subscriber waits for a response from the QuackIt server before sending the next string. If the response is "accept" then the next string can be sent. If the response is "reject" then something went wrong and the publisher should try again or terminate.

- As discussed above, the QuackIt server talks to the publisher and subscriber processes over the pipes and responds to their commands with "accept" or "reject" accordingly. If the *connection protocol* is done successfully, the QuackIt server will create a *connection record* indicating whether the connection is a publisher or subscriber, what specific publisher (pubid) or subscriber (subid) it is, and the pipe associated with the connected process.

- After all of the connections have been made, the QuackIt server prints out the records and begins the *termination protocol.* Here, it waits for each publisher and subscriber to send the command:

  "terminate"

  The QuackIt server responds with "terminate" and then closes the pipe. It prints out a message indicating that which publisher or subscrber was terminated.

The purpose of Part 1 is to get all of the processes that will be operating in your experiments up and running. You will be given a working skeleton of the Part 1 code in lab. In addition, you will be given versions of publisher and subscriber processes to use in your tests.

Part 2 – QuackIt Server Multithreading

Part 1 implements a basic single-threaded QuackIt server for establishing publisher and subscriber connections. Using only a single thread means that it has to handle all of the connections to the publishers and subscribers by itself, in addition to the rest of the QuackIt server pub/sub functions. Part 2 improves on the Part 1 implementation by creating a thread for each connected publisher and subscriber that will operate as a server-side "proxy" for their actions.

Part 2 does the following right after a publisher or subscriber is connected:

- The QuackIt server process creates a proxy thread for each producer and subscriber after all have been connected. The thread is provided with the connection record. If all of the publishers and subscribers are successful in connecting, there should be n proxy publisher threads, T_i^p , $1 \leq i \leq n$, and there should be m proxy subscriber threads, T_j^s , $1 \leq j \leq m$.

- Each T_i^p thread inherits the pipe to its associated P_i publisher. It waits for a "terminate" command from P_i, responds with "terminate" message, and closes the pipe, printing out a message to that affect.

- Each T_j^s thread inherits the pipe to its associated S_j subscriber. It waits for a "terminate" command from S_j, responds with "terminate" message, and closes the pipe, printing out a message to that affect.

You should use Pthreads to implement the server-side threading. Be careful to make the code you develop *thread safe.* Completing this part gives us the functionality of Part 1, but now using thread proxies to handle the connections.

Part 3 – Establishing Topics

Now that we can connect publishers and subscribers to the QuackIt server, we want to let them publish and subscribe to topics. Part 3 implements is the protocol to convey this information to the QuackIt server. Do the following:

- For each publisher T_i^p thread, listen for commands from publisher P_i about what topics it wants to publish to. It will send multiple messages of the form:

  "pub pubid topic k" ($1 \leq k \leq t$, for each topic of interest)

  followed by the message:

"end"

After each message string, the publisher waits for a response from the QuackIt server before sending the next string. If the response is "accept" then the next string can be sent. If the response is "reject" then something went wrong and the publisher should try again or terminate. A separate "pub pubid topic k" message is required for each topic of interest.

- Similarly, for each subscriber T_j^s thread, listen for commands from subscriber S_j about what topics it wants to subscribe to. It will send multiple messages of the form:

  "sub subid topic k" ($1 \leq k \leq n$, for each topic of interest)

  followed by the message:

  "end"

  After each message string, the subscriber waits for a response from the QuackIt server before sending the next string. If the response is "accept" then the next string can be sent. If the response is "reject" then something went wrong and the subscriber should try again or terminate. A separate "sub subid topic k" message is required for each topic of interest.

- The topics information for each publisher and subscriber is added to their *connection record.*
- Once each publisher/subscriber is done indicating their topics of interest, the associated thread should undertake the termination protocol and print out the connection record and return.
- The QuackIt master waits for all threads to return and then exits.

Once this stage is completed, it is time to set up the memory for storing topic messages.

Part 4 – QuackIt Topic Store

Now that we can connect publishers and subscribers to the QuackIt server, you need to implement the functionality to take in a topic message from a publisher and make it available to subscribers. A queue will be created for each topic where new topic entries are saved. The objective of Part 4 is to build the QuackIt *topic store.* Do the following:

- For each published topic, create a circular, FIFO topic queue capable of holding MAXENTRIES topic entries, where each topic entry consists of:

  ```
  struct topicentry {
    int entrynum;
    struct timeval timestamp;
    int pubID;
    char message[QUACKSIZE];
  }
  ```

  There will be $NUMTOPICS$ total topic queues.

- Each topic queue needs to have a *head* and a *tail* pointer. The head of the topic queue points to the oldest entry in the topic queue. The tail of the topic queue points to the last (most recent) entry put in the queue.

- Write an `enqueue()` routine to enqueue a topic entry. Each topic entry enqueued will be assigned a monotonically increasing entry number starting at 1. The topic queue itself will have a entry counter. The `enqueue()` routine will read the counter, increment it, and save it in the topic entry. A timestamp will be also taken using `gettimeofday()` and saved in the topic entry. Subscriber proxy threads call enqueue() to store topic entries.

- Write an `getentry()` routine to get a topic entry from the topic queue. The routine will take an argument `int lastentry` which is the number of the last entry read by this subscriber proxy thread. The routine will attempt to get the lastentry+1 entry, if it is in the topic queue. If this next message is in the queue, `getentry()` will copy it to the `topicentry` structure pointed to by the `struct topicentry *t` argument and return 1. Otherwise, if the topic queue is not empty, `getentry()` will copy the next entry in increasing (newer) order to the `topicentry` structure pointed to by the `struct topicentry *t` argument and return its entry number. If the topic queue is empty, `getentry()` return the negative value of the entry number of the last entry enqueued for this topic.

- Messages can get too old to keep in the topic queues. If a message ages δ beyond when it was inserted into the queue, it is should be dequeued. Write a `dequeue()` routine that dequeues old topic entries starting at the head of the queue. This routine will be executed by only 1 thread (see below).

- Because there can be multiple publishers and subscribers for each topic, access to each topic queue must be synchronized. The `enqueue()`, `getentry()`, and `dequeue()` routines should implement the necessary synchronization.

- Test the QuackIt topic store to show that it is working. At this point, it is ok to just have the producer proxy threads generate a sequence of enqueue calls to topic queues and subscriber proxy threads generate a sequence of dequeue calls.

Figure 2 give a high-level view of the topic entry queueu. There is one of these queues for each topic. It is a fixed size and should be implemented as a circular ring buffer.

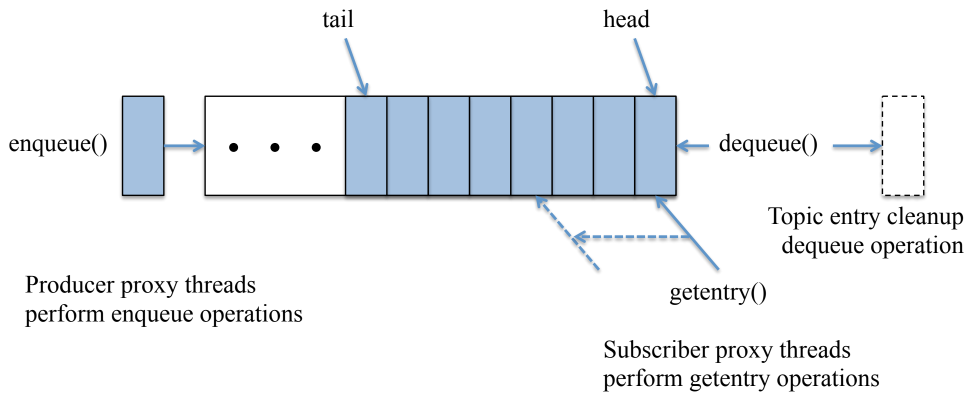
tail
head
enqueue()
dequeue()
Topic entry cleanup dequeue operation
getentry()
Producer proxy threads perform enqueue operations
Subscriber proxy threads perform getentry operations

Figure 2: Topic entry queue and operations.

Part 4 is trickier than it seems. Let's start with the publishers. Any topic can have multiple publishers. Enqueueing must be synchronized, but it is possible for a topic queue to become full. If a publisher wants to enqueue an entry to a full topic queue, it has to wait until there is space to do so. The simplest way to do this is to just yield the CPU and test again when the thread is re-scheduled. (If you want to get fancier, you could consider using a blocking semaphore to implement this.) Eventually, a dequeue operation will come along and free up queue space.

Now consider the subscribers. All subscribers for a topic must read topic entries in order, using a monotonically increasing message number. However, once a message has reached a certain age, it might be dequeued instead of read by a subscriber. Who does the dequeuing? The `getentry()` routine will only indicate whether the next entry is available or, if not, whether there is a newer entry available. The problem is that either there are no newer entries for this particular subscriber proxy thread or the queue is empty. In either case, the subscriber proxy thread should try again. The simplest way to do this is to just yield the CPU and try again when the thread is re-scheduled. (If you want to get fancier, you could consider using a blocking semaphore to implement this.) Eventually, an enqueue operation will come along.

A separate *topic cleanup thread* will be used to periodically call the `dequeue()` routine on every topic to get rid of old messages.

Part 5 – QuackIt Topic Archiving (EXTRA CREDIT)

By putting the QuackIt topic store in memory and having multi-threaded access, you are creating a high-performance QuackIt server solution that could be run on a multiprocessor shared-memory machine and be accessed by many publishers and subscribers. However, the main memory of that machine is finite and you have to be concerned about what to do when a topic queue fills up. You can make MAXENTRIES large, but that does not solve the problem. It is important to always have space to bring in new news. It is equally important to save a complete history of every topic entry published. The solution is to store old news offline in a QuackIt *topic archive*. The real question is when does an entry migrate to the topic archive.

This is where the *timestamp* comes in. The basic idea is to have a limit on the amount of time (δ) an entry can be in the topic store before it has to be migrated to the topic archive. In Part 4, we discussed how an old topic message can be detected by a topic cleanup thread. Instead of justs deleting the topic entry, this is a great time to archive it. However, it is not a great idea to do the I/O right inside the `dequeue()` routine. Instead, a separate *topic archiving thread* could be used for this purpose, with a double buffering approach. Do the following:

- For each topic, create 2 FIFO buffers, each able to hold IOBUFFERSIZE (e.g., IOBUFFERSIZE = 1000) topic entries. The idea is that a buffer will be filled with old entries for archiving until it is full. At that point, it will be written to the topic archive file. The other buffer now will be filled while the full buffer is written.

- Modify the `dequeue()` routine to archive old entries. If the entry is old, call `insert()` to insert it in the topic archive buffer and then dequeue it.

- The `insert()` routine will signal an *archiving thread* if the buffer becomes full and then switch to the other buffer. The archiving thread will keep track of when buffers are full, and write them to the topic archive file. The archiving thread should create a file for each topic when it starts up that will serve as the topic archive.

The technique of using 2 buffers where one is being filled while I/O is taking place on the other is called *double buffering*.

Part 6 – QuackIt Publishing and Subscribing

TBD.

Part 7 – QuackIt Goes Live!

TBD.

Developing Your Code

The best way to develop your code is in Linux running inside the virtual machine image provided to you. This way, if you crash the system, it is straightforward to restart. This also gives you the benefit of taking snapshots of system state right before you do something potentially risky or hazardous, so that if something goes horribly awry you can easily roll back to a safe state. You may use the room 100 machines to run the Linux VM image within a Virtualbox environment, or run natively or within a VM on your own personal machine. Importantly, do not use `ix` for this assignment.

Individual Work and Helping Classmates

This is an individual assignment. You all should be reading the manuals, hunting for information, and learning those things that enable you to do the project. However, it is important for everyone to make progress and hopefully obtain the same level of knowledge by the project's end. If you get stuck, seek out help to get unstuck. Sometimes just having another pair of eyes looking at your code is all you need. It is understood that students have different levels of programming skills. If you can not get help from the TA, it is possible that a classmate can be of assistance.

Grading

The grading will be based on the project as a whole. Credit will be given for parts completed, but the score will be weighted more towards full solutions. You should make sure to be able to demonstrate what you have working.

The project is due on November 30 at midnight. That gives 4+ weeks to complete the work. 10% will be deducted for each day (or portion thereof) late. Only 2 late days will be allowed, after which no credit will be given.