Project #2 - Around*the*Pond

CIS 415 - Operating Systems
Fall 2015 - Prof. Malony

Due date: December 4, 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 version that would appeal to the UO undergraduates that make up his fan base. After consulting his "branding" manager, Puddles is calling his new *app* **Around*the*Pond)** (ATP)! 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, ATP will have proprietary technology called *Quacker* 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.

AroundthePond must be responsive, scalable, and rival anything that can be found at other Pac-12 schools. You have an idea about how to use a pub/sub framework to develop the Quacker-based technology needed for the ATP social networking service. With the extraordinary skillset that you are learning in CIS 415, you decide to implement the pub/sub architecture shown in Figure 1). Your objective is to create a solution that will run faster than anything Oregon State University could ever imagine. Besides, **Around*the*Dam** just does not have the same ring to it.

There are 6 parts to the project, each building on the other. The objective is to demonstrate a combination of OS techniques in your solution: interprocess communication, threading, synchronization, and file I/O.

Part 1 – Connecting to AroundthePond

The goal of Part 1 is to develop the first version of the AroundthePond 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 the ATP server 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 ATP server process, n publisher processes ($P_i,\ 1 \leq i \leq n$), and m subscriber processes ($S_j,\ 1 \leq j \leq n$).

- The ATP 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 and has established a pipe with the ATP server, it send the following messges:

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

  After each message string, the publisher waits for a response from the ATP 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.
- For each subscriber, S_i, send the following messages to the ATP server:

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

  Again, after each string, wait for a response from the ATP 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 "sub subid topic k" command string is required for each topic of interest.
- As discussed above, the ATP 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 ATP server will create a *connection record* indicating whether the connection is a publisher or subscriber, what specific publisher or subscriber it is, what topics were selected, and the pipe associated with the connected process.
- After all of the connections have been made, the ATP server prints out the records and begins the *termination protocol*. Here, it waits for each publisher and subscriber to send the command:

  "terminate"

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

Part 2 – AroundthePond Server Multithreading

Part 1 implements a basic single-threaded ATP server for establishing publisher and subscriber connections. If you continued to have only a single thread, it would have to service all of the connections to the publishers and subscribers by itself, in addition to the rest of the ATP 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 ATP server process creates a proxy thread and provides the thread 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 , $1 \leq j \leq m$.

- For each T_i^p thread, wait for the associated P_i publisher to send a "terminate" command and then respond with "terminate" and close the pipe, printing out a message to that affect. For each T_j^s thread, wait for the associated S_j subscriber to send a "terminate" command and then respond with "terminate" and close the pipe, printing out a message to that affect.

Pthreads will be used to implement the server-side threading. Be careful to make the code you develop *thread safe*.

Part 3 – AroundthePond Topic Store

Now that we can connect publishers and subscribers to the ATP server, you need to implement the functionality to take in topic information from a publisher and transfer it to a subscriber. Because there might not be a subscriber ready to take topic information, it has to be stored until the subscriber is ready. Also, there may be multiple subscribers interested in the topic. The objective of Part 3 is to build the ATP *topic store*. Do the following:

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

  ```
  int timestamp;
  int pubID;
  int data[ENTRYSIZE];
  ```

  There will be t 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 entry put in the queue.

- Write an *enqueue()* routine to enqueue a topic entry and a *dequeue()* routine to dequeue a topic entry. Because there can be multiple publishers and subscribers for each topic, access to each topic queue must be synchronized. The *enqueue()* and *dequeue()* routines should implement this synchronization. (Timestamps are taken within the *enqueue()* operation.

- Test the ATP 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 and subscriber proxy threads generate a sequence of dequeue calls.

Part 3 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 be 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. This will happen either when all subscribers have read the entry at the head of the topic queue, or when entries are archived (see Part 4). Now consider the subscribers. All subscribers for a topic must read all topic entries (unless the entry has been archived). Thus, each subscriber must have its own pointer to the next entry to "dequeue" from the topic queue. (When a subscriber is created, its entry pointer will be set to the topic queue head pointer.) The tricky part is knowing that all subscribers have performed the dequeue for each entry. With this in mind, be careful in your implementation of the *enqueueu()* and *dequeueu()* routines.

Part 4 – AroundthePond Topic Archiving

By putting the ATP topic store in memory and having multi-threaded access, you are creating a high-performance ATP 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. Given that it is important to save a complete history of every topic entry published, the solution is to eventually store old news offline in a AroundthePond *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 an entry can be in the topic store before it has to be migrated to the topic archive. The question is when to check this. If a subscriber attempts to dequeue an entry from a topic queue and the entry is too old, the AroundthePond design requirements state that it is ok to ignore the entry. If this is the last subscriber to dequeue the entry and it is old, that is a great time to archive it. However, this solution relies on every subscriber for the topic to be requesting an entry in a timely manner. Actually, this is not a problem if you keep in mind that each subscriber has a subscriber proxy thread. Think about this.

If an archive action needs to occur, it is not a great idea to do the I/O right inside the *dequeue()* routine. Instead, you decide to use a separate topic archive thread and a double buffering approach. Do the following:

- For each topic, create 2 FIFO buffers, each able to hold IOBUFFERSIZE (e.g., IOBUFFERSIZE = 100) topic entries. The idea is that a buffer will be filled with old entries for archiving until it is full. At that point, the other buffer will be filled while the full buffer is written to the topic archive file.

- Modify the *dequeue()* routine to detect old entries. If the entry is old and this is not the last subscriber proxy to dequeue the entry, just advance to the next entry. If the entry is old and this is the last subscriber proxy to dequeue the entry, insert the entry into the topic buffer being filled by calling the *append()* routine and advance to the next entry.

- Create a *archiving thread* whose responsibility it is to 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*. It is in the *append()* routine that the buffer filling and switching is taking place. It a buffer is filled up, the archiving thread needs to be informed to do the I/O.

Part 5 – AroundthePond Publishing and Subscribing

Publishing and subscribing to AroundthePond will require a protocol to be established. Publishers and subscribers can connect and terminate. Now we need that part of the protocol to publish an entry or obtain an entry for a topic.

Let's start with publishing. A publisher can publish on multiple topics. It can be assumed that publishing an entry for a particular topic will be somewhat responsive, that is unless the topic store is full. In this case, if the publisher proxy blocked, it might be that the published wants to send in entries for other topics whose topic store are not full. To address this problem, do the following:

- If a publisher, P_i, wants to publish an entry for topic k, it sends the following to its proxy:

  "topic k [entry]"

  where [entry] is ENTRYSIZE worth of bytes.

- The publisher's proxy reads the message. If it was successful in publishing the last entry for this topic, it saves the entry into a temporary entry data structure and responds:

  "successful"

  If it was not successful in publishing the last entry for this topic, the proxy responds:

  "retry"

- Once the publisher proxy has accepted the entry, it then goes on to try to enter it into the topic store. Note, there needs to be a temporary entry in the proxy for each topic of interest to the publisher.

The process above will let the publisher proxy be responsive to all topics of interest, but places the burden of retrying back on the publisher. Puddles thought this was an acceptable compromise.

Now let's look at getting topic entries in the subscriber. After registering with the ATP server, the subscriber's proxy knows the topics of interest. Its responsibility then is to find the next entry for each topic of interest and send it to the subscriber. It does so as follows:

- If the subscriber proxy, T_k^s, sees that there is an entry for topic k in the topic store, it checks to see if the entry is old. If so, it follows the procedure above and moves to the next entry. If the entry is not old, it reads the entry into a temporary entry and then sends the following to the subscriber:

  "topic k [entry]"

  where [entry] is ENTRYSIZE worth of bytes.

- The subscriber upon receiving the message, extracts the [entry] payload and responds to the proxy:

  "successful"

- Once the communication is successfull, the proxy moves to the next topic of interest.

The process above will let the subscriber proxy be responsive to all topics of interest, but it places the burden on the subscriber for successful communication. This is a vulnerability, since it is possible that the subscriber never responds. Ideally, what you might want is for the subscriber proxy to set a timer, after which it assumes the connection to the subscriber is dead.

Part 6 – AroundthePond Goes Live!

All of the pieces of AroundthePond are now ready to put together and demonstrated. It can be tricky to test everything out in such a dynamic, concurrent architecture. Part 6 is to do so in a systematic way. The following experiments are suggested:

- **1 pub / 1 sub / 1 shared topic:** This will test the flow through the ATP system. You can vary the rates of the publisher and the responsiveness of the subscriber to cause boundary cases to occur.

- **1 pub / m sub / 1 shared topic:** This will further test the flow through the ATP system with emphasis on subscriber functionality.

- **n pub / 1 sub / 1 shared topic:** This will further test the flow through the ATP system with emphasis on producer functionality.

- Run the test above with 2 topics.
- n **pub / m sub / t shared topic:** If you got the other tests working, this parameterized test is sure to make things break.

The goal here is to cover as many scenarios as possible with you experiments. While the actual entries do not make, you might make think about making an entry meaningful in some way, like a string containing some distinguishable set of characters (e.g., pub ID, entry ID). Also, other parameters, like MAXENTRIES and how old an entry has to be before being written to the topic store, you should set judiciously.

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.

Please use your Bitbucket GIT repositories for keeping track of your programming work. Please use a *makefile* and the *make* utility for building your program. This will be of benefit to you and will help in the grading.

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 Dec. 4 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.

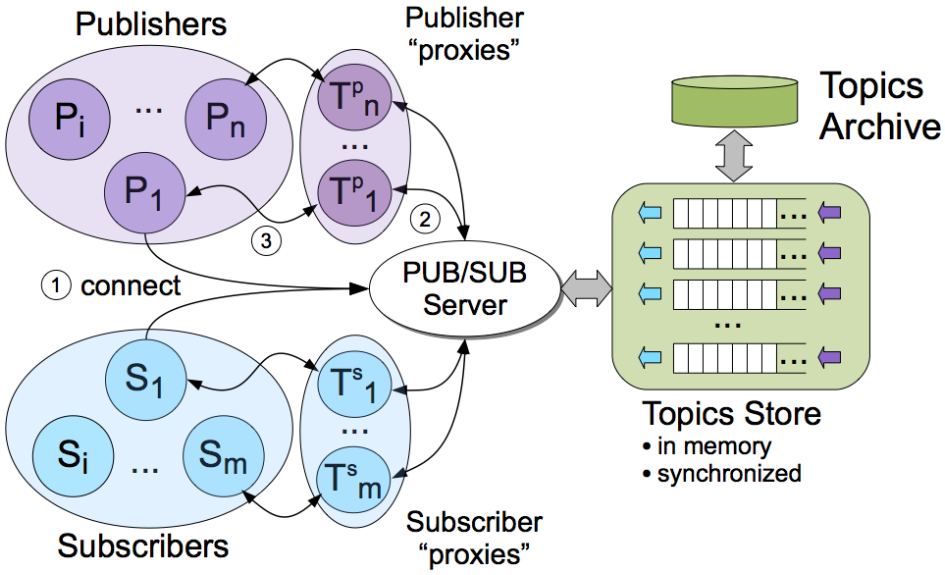

Figure 1: Pub/Sub architecture for the Quacker-based ATP server.