Robert Larkin, CSCE689, HW4 Replication Clients

Date 2/28/2020:1600hrs, Due 2/28/2020:1700hrs

GITHUB: <https://github.com/larkinrd/AFIT-CSCE689-HW4-S>

Note: Unless specified otherwise, all references are to our course textbook.

**Introduction**: My server architecture does not vary that much from what was provided by Lt Col Noel. I did majority of my coding inside the TCPConn object and leveraged the ability to exchange information from one server to the other through communications. This includes replicating data and relaying timing information so that servers can decide on a master clock and how to keep data being stored and replicated consistent. I was able to complete the authentication, encryption, and a portion of the time skew issues. I believe my time skew is solved through my NTP type communications, but I was not able to fix data consistency as I had trouble overwriting each servers individual drone plot timestamp to the newly determined master clock. This document addresses each of the major requirements of consistency, communications, naming convention, coordination techniques, and provides a critique of the supplied HW4 architecture. On pages 3 and 4, it ends with a summary of the HW4 objectives, a recap of the ROE’s for this assignment, and more detailed on answers on timing, synchronization, and data consistency as it applies to how I approached the HW4 problem set.

**Consistency model**: The desired consistency models for my architecture is a combination of sequential and concurrent consistency. It is sequentially consistent within the server where multiple processes are sharing a data store (.e.g., its DronePlotDB). The operations/functions to add, sort, replicate, and deduplicate data within the DronePlotDB datastore need to be sequentially consistent such that the result of any execution by a process is the same (i.e. predictable). For adding DronePlotDB information, a write operation, it should use *read my writes* consistency to ensure subsequent reads by that process see the same result. The add operation can also be concurrent with another process because the server can add information from its own drone/antenna and another process receiving replication data can add it to the data store. We just need to ensure that entries going to the data store have their own space to store the entries so they are not clobbering the exact same data locations. The other operations sort, replicate, and deduplicate involve both reading and writing to the data store and I would choose a *write follow reads* model to ensure that the the process(es) do not read an ‘old’ value ensuring it has the most up to date value when moving data to a different storage location. This also applies to replication because we are writing to flags and changing data. We do not need causally consistent because not all information received/transmitted needs to be seen in the same order by the individual servers.

**Communications method**: We have connection oriented communication much like figure 4.19, pg 195 where the server establishes a listening port ready to accept inbound connections. Likewise, our server will initiate outbound connections through a random high numbered ephemeral port where it connects to the other servers listening socket and sends information. Our queue manager, in charge of setting up TCP connections, is not limited to sending one DronePlotDB point at a time. The queue manager buffers all the data it has received marshalling by the ReplServer.cpp from the DronePlotDB over a 20 second time period and prepares it for transport to the other replication servers (much like the Message Passing Interface (MPI) described in Figure 4.25, pg204. In our architecture, the MPI is more similar to QueueMgr.cpp which is a child of a TCPServer for setting up and tearing down connections, whereas ReplManager is more likened to the basic interface which performs put, get, poll, and notify operations described in Figure 4.27, pg 208. To transfer the message, our architecture employs the attributes of a message channel agent described in Figure 4.31, pg 215 where our QueueMgr.cpp sets the transport type to TCP, delivery is First In First Out (FIFO), message length is dynamically allocated depending on how many drone plots we are replicating, and I believe both the setup retry count is 2 and delivery retry is 2. If we were to draw an overlay network, it would show a simple network, with only our servers, in which they communicate via broadcast sending all communications to each other. We are not multi-casting because we are not relaying information to a subset of the known servers. Moreover, we are not gossiping because we are not relaying information to a subset of nodes and we know all the other nodes via our servers.txt file, nor are we employing epidemic protocols for the same reasons. Also, our implementation does not suffer from high link stress, high relative delay, or high tree cost because all servers are local and resident on the same machine (i.e. we are not operating over a true network).

**Naming convention**: We are using a flat-naming convention in our architecture. Our architecture is omniscient and knows all other servers and IPs via a servers.txt file that hosts the servers name, IP address, and its listening port. We employ a hierarchical naming convention within our C++ class objects in which we have child and base classes where QueueMgr is a child of TCPServer and TCPServer is a child of Server. Lastly, you could say our architecture employs forwarding pointers via the use of iterators and queue structures, but I believe this to be a stretch, especially given the example on pg 244 and knowledge of how cell phone networks operate (you have a home of record and your temporary/transient location in which forwarding does occur to get clients the right information). Our program doesn’t really forward, but rather traverses up and down various data structures in a very manual fashion. Similarly, when we deserialize the DronePlot database, we traverse through a record and index into the part we want (such as timestamp) and call it directly through C/C++ pointers. With respect to the rest of the material presented in our text book, we don’t allocate home entities, nor do we have a structured peer-to-peer system to route messages through one node to another.

**Coordination techniques**: I will explain in more detail my thinking and rationale in the summary section on pages 3-4 as I address questions posed on the HW4 canvas site. I used an NTP type approach to exchange timestamps from one server to the other. In our architecture, we don’t really have to account for time delay since everything is resident on the same machine. In the real-world, we have to account for the amount of delay for communications between two servers. For design purposes, we must determine how accurate of a time measurement do we need. I argue that for Drones delivering packages, having accurate timestamps within a few seconds (1-5) is probably good enough. The major factor would be how fast drones travel, if they travel a few meters/sec, then a 1-5 second timestamp is good enough to show operators the drones approximate location. For precisely dropping a bomb and adjusting its trajectory, we need near real time timing. For our architecture, we could even employ a underwater radar type screen (think incoming torpedo where the dot updates every few seconds) where we see drones on a map that refresh every five seconds. By refreshing every five seconds, we give our servers more processing time to do calculations on DronePlotDB entries, deduplication, database cleanup, and more. For my election algorithm, I simply chose the system clock with the lowest timestamp, this way servers are not writing information into their data store in the future. One thing missing from my architecture is the ability to routinely determine who has the lowest clock and update my offset accordingly. I would also want to have the ability to trigger a new election when a new server joins the pool of servers. In the case that server has a lower timestamp, I’d probably not allow him to join the group until he has gotten his clock correct to match the server with the current lowest time (this assumes we have a correct master clock). One last topic is that of mutual exclusion, while troubleshooting with Lt Col Noel, we needed certain processes to fire off sequentially and an easy solution was to identify data that was dependent on other threads and essentially block other processes until the data they need is ready through the use of a mutex. Using a mutex can also apply to strengthen consistency models discussed above where we need to determine what are our critical data regions and protect it from corruption/inconsistency. Keeping with reality, our drone package delivery system can suffer loss or corruption of one data point and probably survive the loss of a minute or two not knowing what is going on with a drone. However, I’d argue that after two minutes of lost/corrupted data, our boss will want to know what is going on with the software and want answers about where drones and packages are.

**Critique the existing server architecture and indicate any areas you would change**

* I would not try to hide how the drones, antennas, and servers get their startup time. With a little bit of time, we have all the source, we can figure it out anyway.
* I would not use negative time for servers starting up.
* One interesting situation occurred when I delayed (through my bash script) that startup of one server by 15 seconds. My NTP type algorithm does not account for that.
* Emphasize the use of wireshark within the class, it was really helpful for me to see if packets were sent or received or stuck in a buffer somewhere. It also shows what is and is not encrypted.
* I would provide students a basic bash script which resembles how you will start our code. Are you going to start all three servers at the same time, delay each by 1 second, ???
* I would rename and/or reorganize ReplServer and QueueMgr. Per the book, a server is a process implementing a specific service on behalf of a collection of clients. As you stated, we have many overlapping terms. I propose you swap the terms and change ReplServer to ReplicationManager as it is reading from the drone plot manager and sending info to the QueueingServer which runs a service to send/receive information. I was always playing with the idea of using the terms service, daemon, and object server… but they didn’t quite fit my liking.
* I would also provide more details on how to interact with DronePlotDB, but this could be my lack of familiarity with C++ syntax

**(A summary of HW4 Objectives and answer to questions on the HW4 Canvas site are on the next page)**

**SUMMARY OF HW4 MAJOR OBJECTIVES COMPLETION**

1) Provide authentication – COMPLETE

2) Provide encryption – COMPLETE

3) Deduplicate plots – ATTEMPTED/PARTIALLY COMPLETE

4) Adjust for time skew – COMPLETE, but depends on implementing Deduplication

**HW4 RULES OF ENGAGEMENT – Items of no interest are in gray font**

* *Code to simulate receiving data from the drones (operates in its own thread; do not modify this code)*
* *APIs via the DroneDB class to interface with the saved drone data - you can make some modifications to DronePlotDB but it should not break the simulator or "break the fourth wall".*
* *As usual, do not change the main.cpp file*
* *The DroneDB class will have a data dump function. A "diff" command between your output should show no differences except the "answer solution" can be slightly off its timestamp as long as the deviation is the same throughout (i.e. consistency is maintained) and the "answer solution" can have slight variations on node\_id where duplicate entries were found*
* *Each new connection should begin with a challenge/response handshake using randomly generated bits and reject connections that fail to properly encrypt the bits in response - use the process defined in Figure 9.6 of your text (pg. 514)*
* **ADDED**: Avoid using global variables. Follow software design and coding best practices. **NOTE**: while the use of global variables is sloppy it proved useful in debugging and troubleshooting timing issues between the drone, antenna, and server. I found it to be the easiest to pass data back and forth between ReplServer.cpp and TCPConn.cpp. A more correct way is to pass in the reference of a ReplServer into certain TCPConn functions for this assignment.

**HW4 QUESTIONS/STATEMENTS FROM CANVAS HW4 ASSIGNMENT PAGE**

* **System clocks may not be synchronized to the second;  data at two sites may have identical location readings that are off by a second or two. If you write both of those to the database, it will look like the drone stopped suddenly for a few seconds. How will you reconcile this?**
  + I would use two methods to resolve this issue. First, as illustrated in my code, I would develop a Network Time Protocol type solution with an election based algorithm such that servers periodically communicate to determine who has the master clock, what is the master clocks’ time, and by what amount do I need to offset my local time to be synced with the master clock. Second, since data storage is cheap, I would retain all of my timestamps from various devices in the network. Beginning with the Drone, I would have the drone send its timestamp when it transmits to any antenna’s within the area. Assuming that transmission takes less than one second to travel from the drone to the antenna, I can have all of my timestamps rounded to the nearest second. Next, if my server is collocated with my antenna, my antenna can be dumb and provide no timestamp. If my antenna is smart, it can add its timestamp at multiple layers of the OSI model. The most logical choice is to incorporate the antenna into our middleware design such that it has the capability to append its own timestamp, with its own <AntennaTime> tag, in the data portion of a TCP segment at the application layer. If that is not possible, my next choice would be to insert an Option-Kind 8, TCP Timestamps Option, within the TCP Segment header (see RFC 1323, <http://www.networksorcery.com/enp/rfc/rfc1323.txt> and <http://www.networksorcery.com/enp/protocol/tcp/option008.htm>). The reason for keeping all my timestamps is so that I can figure out if something is wrong with a drone, antenna, or server. Plus I can perform forensics or other data analytics. Lastly, and especially if I have no control of the antenna and/or decided its not really important to collect an antenna’s timestamp, I will have to figure out the amount of time delay between a Drone broadcast, Antenna receipt, and time for that transmission to arrive at my server. This goes back to asking the question, will I trust the time generated by my Drone, or simply trust the server and overwrite a drone’s timestamp with the servers timestamp.
* **Similarly, due to atmospheric variations, two sites may receive data intermittently, making it look like the drone is jumping back and forth if their clocks are not sync'd.** 
  + This situation is remedied when the clocks of two servers are known to be synced because while the timestamp sent by the Drone will be the same, each server will overwrite that timestamp, but you can use the the location data to determine a duplicate because drone is not going to be in ‘exact*’* same spot. When location data is ‘exactly’ the same, we’ll use the server with the minimum time since we know the transmission was delayed from the drone to the other server in this case. This can be solved with a deduplicate() database routine that runs periodically to remove duplicates. If business practices for my company dictate that drones should be moving (i.e. not idle) then we only need GPS coordinates down to the fifth (+/- 1.1m) or sixth (+/- 0.11m) decimal place. If my drone is not moving at least 1.1m within 5 seconds (to include changing altitude), my drone must be at the location putting the package on the ground moving slowly, or something else is happening to my drone for which an alert would be triggered to get the attention of an operator. Since time is money, a slow moving drone will make the company less money and this is undesirable in our system.
* **You are unable to directly synchronize the system clocks** 
  + I assume this meant no cheating, no setting/programming clocks to what you want for this assignment. My approach leveraged network communications to transmit my servers clock system time to the other servers; however, it was challenging to obtain a predictable and repeatable system startup times as they related to the drone/antenna random offset. This led to troubleshooting and bug hunting with Lt Col Noel to get Drone, Antenna, and Server timing configured as he had intended.
* **Access to the database will be mutex'd between your thread and the simulation thread (N/A)**
* **It is up to you to define your naming convention and communications methods, but they should be efficient and reliable. Many methods already exist within the code that can greatly simplify this task.**
  + I took advantage of the ability to create my own XML style tags and transmit those to the other replication servers. A summary of my XML tags are below
    - <T0><T1><T2> - Three opportunities to transmit/receive timestamps. This results in six packets going across the wire to build a vector table of all other servers SID and Timestamps. Due to the design of the system, and assuming all servers are online with different clocks, the ‘first’ server WILL transmit to ALL other servers and receive responses. This allows all servers to communicate, record the other servers timestamps, compare their timestamp, determine who has the minimum time, and figure out their offset. The server with the minimum time was chose as the master time.
    - <CHAL> - The initiating server sends an **UN**encrypted challenge with a random string. To prevent the same random challenge from being generated, the initiating client (the server sending his data) issues a challenge string 10 random characters and the server receiving the data issues a challenge string of 20 random characters.
    - <RESP> - The server responds to the Challenge with an **EN**crypted responses
    - <SID> - Represents the servers identification. Note, to set the receiving servers SID passed back to the initiating client, within TCPServer.cpp::handleSocket() I dynamically cast ‘this’ TCPServer receiving the connection to a child object of QueueMgr and setSvrID. Although fixing this bug was not necessary, it proved useful to know exactly which IP and which SID was sending which communication when viewing wireshark captures with 3 or more replication servers.