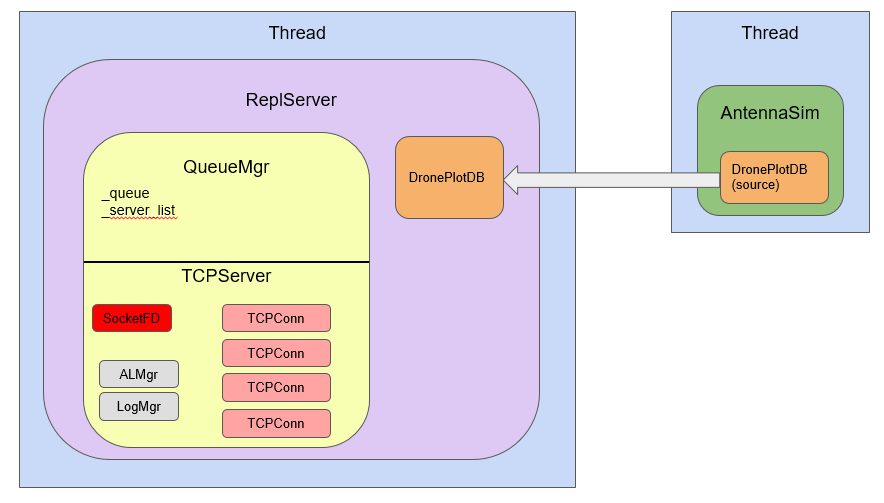
# CSCE 689 - Homework 4 Design Document

## Overall Architecture



Homework 4 architecture consists of two main elements. The AntennaSim class operating in its own thread reads from a source database and injects DronePlot objects into the replication server at the appropriate time. The speed of the “clock” can be sped up using the command line parameter “-t” where “-t 2.0” would operate twice as fast”. This can be useful to quickly run through the 800 seconds of the simulation when debugging.

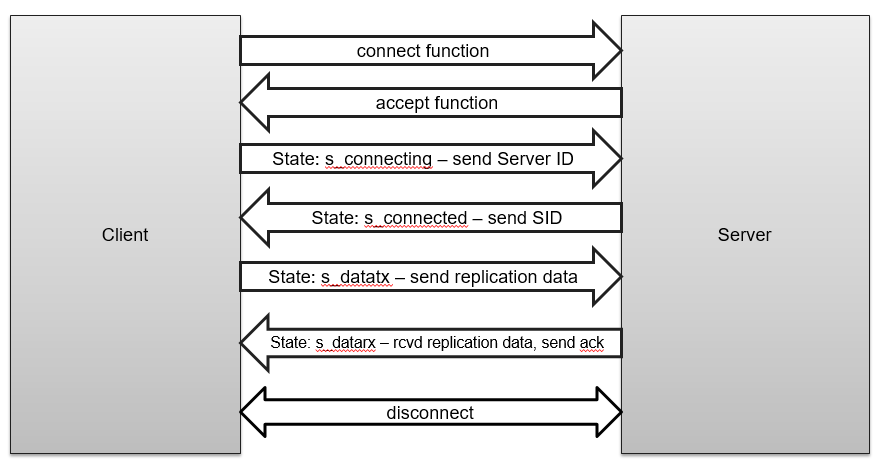
Drones broadcast their GPS coordinates every 5 seconds. The clocks of the receiving nodes are not completely sync’d and could be up to six seconds off. (In reality, this is simulated by a randomly-generated offset of -3 to 3 seconds when the server starts, so it will be different every time). The drone plots will be fed by the simulation thread into the DronePlotDB class object managed by the ReplServer class object.

The ReplServer::replicate() function loops infinitely until ReplServer::shutdown() is called. The primary loop does the following actions:

* Before the loop begins, the listening socket is established
* While not shutting down
  + Call \_queue.handleQueue that performs routine queue maintenance. This includes:
    - Checks the socket for new connections and, if found, accepts adds a TCPConn object to the connection list
    - Loops through all connections, receiving and sending data depending on the “status” or “state” of the connection
    - Loops through connections one last time and places any received data into the queue to be “popped”, or accessed, later (FIFO)
  + If it has been 20 simulation seconds since the last replication, loops through the database looking for “new” plots (identified by the DBFLAG\_NEW flag) and marshalls them into a serialized message, adding it to the outgoing queue
  + Empties the queue, creating TCPConn objects for outgoing data and replicating in received data to the database

The design is intended to be an object-based design. To communicate with another server, all the application needs to do is place a message in the queue with a ServerID target (“DS1”, “DS2”, “DS3”, “BobServer”, etc). The lookup table is managed in QueueMgr (\_server\_list) and all communications happens automatically. The ReplServer merely needs to check the queue periodically for a reply. In this way, QueueMgr could be used for a myriad of different asynchronous applications. While the QueueMgr is concerned with asynchronous communications, the ReplServer is focused on replicating and deconflicting data to provide via API to the theoretical “Application Layer”.

## Lower-Level Communications



Lower-level communications is encapsulated in the TCPConn class object. Messages are bracketed by start and end tags (similar to XML). So replication data may be surrounded by <REP>data data data data</REP>. Functions have been provided to largely simplify managing tags:

* wrapCmd – places the start and end tags provided in the params around the data
* getCmdData – removes the start and end tags and returns the data in the middle
* hasCmd – Simply looks for a tag and returns boolean if found
* findCmd – returns an interator to the position of the tag

The connect function sets the state to s\_connecting and calls the SocketFD connect function and that’s it. From there, handleConnection calls the appropriate function given the current connection state.

The connection is not very secure--there is no authentication and no encryption. Students should add shared key authentication as defined in the course textbook, Figure 9.6, page 514. Keep in mind Note 9.2 starting on the same page to avoid a reflection attack. Also, be sure to pay attention to which steps are encrypted and which are not. For instance, step 1 has Alice send an unencrypted request to connect to Bob—this could be as simple as sending her server ID (as the code currently does). Bob is then going to send an UNENCRYPTED random number string which Alice will encrypt and send back. Not getting this right will defeat the point of the authentication. You will need to add to/adjust the statustype enum to include your additional stages of the connection. Also, you will need to modify/create the functions like sendSID(), waitForData(), etc. so each stage is handled properly.

Functions for sending data include:

* sendData – simple send of unencrypted (or previously-encrypted) data
* getData – gets data without decrypting
* sendEncryptedData – first encrypts the data, then sends with sendData
* getEncryptedData – first calls getData, then decrypts the data
* encryptData – just encrypts a buffer, does not send
* decryptData – decrypts a buffer

Encryption is 128 bit AES block encryption using a shared key stored in sharedkey.bin, which can be generated using the command “./src/keygen sharedkey.bin”. Encrypted data starts with a 128 bit Initialization Vector (IV) and is followed by 128 bit blocks, so it won’t be the same length as the data you passed into the encryption function. Keep this in mind when processing your data on both ends. As long as you pass into decryptData the exact string that was returned from encryptData, it should decrypt just fine.

## The Drone Plot Database

The DronePlotDB class object manages a list of DronePlot class objects. Their definitions can both be found in DronePlotDB.h. Your primary functions will be:

sortByTime – re-sorts the database by timestamp—useful if you’ve adjusted timestamps

begin() – returns an iterator that can be used to move through the list, reading/changing attributes

end() – iterator pointing to the end of the list

erase – removes an item based on iterator or index

popFront – removes the first DroneDB item

addPlot – adds a new drone plot to the database with the given attributes

Replication happens when the server loops through the DronePlot objects and finds the ones with the DBFLAG\_NEW flag set. There are other flags you can use simply by renaming them at the top of the header. They use a bitwise and/or on an unsigned short, so don’t change the hex in the define (i.e. 0x8 should stay 0x8, but you can change DBFLAG\_USER2).

## Deconflicting Replication

Current DronePlot objects are generated by a simulation or replicated in from other servers. There are guaranteed to be duplicate drone plots with potentially skewed times. The application using this replication middleware expects the drone timestamps to be within 15 seconds of the global clock (this is guaranteed by server sysadmin). They do not care where it falls within that 15 second range on either side of global time. They do, however, want all timestamps to be identical between replication servers and be consistent. This means that all timestamps should be approximately 5 seconds apart, even as the drone transitions from one antenna range to another. You will need to develop/implement the following scheme:

1. Identify and implement an election algorithm – recommend keeping it as simple as possible
2. Identify a technique to detect skew and adjust all drone plots

You will need to identify all constraints required for your algorithm to work within your paper.

## Running/testing the servers

Servers can be started using the “./src/repsvr” executable. Command line parameters include:

Command line options:

-a <address>: the IP address to bind the server (default: 127.0.0.1)

-p <port>: the port to bind the server to (default: 9999)

-t <float>: the time multiplier to speed up or slow down the server (2.0 = twice as fast)

-o <filename>: specifies the output file for the CSV database dump when complete

-d <int>: duration to run the server before dumping to the CSV file (default: 800 secs)

-v <level>: verbosity level for debugging info (0-3) – 3 == lots of screen spam

The command line is: ./src/repsvr <options> <binary drone plot file>

The binary drone plot file is provided in multiple formats. First, there is a single drone that leaves the warehouse, flies to a house, then returns. It spends most of its time in Node 3 but starts out in Node 1 and 2. Each file is broken out by node and should be assigned to its own server. Next,there is a ThreeDrone version that includes the same drone, but also includes two more going in different directions. The code will be graded using another version that uses three drones and will not be released.

Servers in the replication scheme are designated by servers.txt (hardcoded filename) and it includes a comma-separated list of parameters. Current format:

<servername>, <ip address>, <port>

Servername can be any name. The QueueMgr will expect to see its own IP address and port in this list—it will exit if it does not. You should put all replication servers in this list—you can run as many as you want, or as few as two. The code will be graded by running three servers and comparing the results of the output.

As an example, I might put two servers in the servers.txt file. Both would have the same IP address but they would each get their own port. I would open two terminals and run the following in each:

Term1: ./src/repsvr –p 9999 –t 2.0 –o svr1.txt –v 3 ./data/SingleDroneN1.bin

Term2: ./src/repsvr -p 9998 –t 2.0 –o svr3.txt –v 3 ./data/SingleDroneN3.bin

This would launch two servers, one on port 9999 and the other on port 9998. I happened to use the node 1 and node 3 versions of the SingleDrone simulation. Most of the plots are in Node 3 so will replicate over to Node 1. It is going to execute at twice the speed, so should take 400 seconds to run. Verbosity is turned all the way up, so it will spam heavily. The results will be written to svr1.txt and svr3.txt.

There will be a three second delay before the simulation starts to allow servers to come online, so you’ll have to start each server within that window.

## Thread Safety

There are two primary threads running—the simulation thread and the replication thread. The DronePlotDB class has mutexes for adding or removing drone plots. The simulation thread will only add new DronePlots, never remove or modify them. Modifying an existing Drone Plot should be atomic and, therefore, perfectly safe. Iterators are provided for accessing the DronePlot objects.