# Authentication:

One of the major parts of the project that was not completed by the grad student was a failure to authenticate connections. In order to improve the security in this manner, a shared key authentication system was used. First the key was put in a place where both the client, or entity connecting, and the server, the entity the client connected to, could access. In the specific file system that was in the main homework directory. The additional security was coded in TCPConn. The system uses a state machine to track what state it is in and what it should do in that state. Before the authentication model was used, there were 7 states. In order to perform the security algorithm, 6 states were added. The algorithm is found on page 514, in *Distributed Systems* (3rd ed), the class textbook. The first step of the algorithm is that the client connects and allows the server to know who is contacting it. In the implementation, that would be the SID transfer. However, since the connection is already established and we were given a code base to start with, I chose to start with the server sending a challenge and the client waiting for a challenge. This allows the code that already exists to still function with almost no modification. Additionally, there is a security benefit of less information about each side of the connection being shared before identities are confirmed. That being said, the server sends a challenge. For challenges, I coded a random generator that selects 16 random numbers from 0 to 255. The odds of that being the same between runs is small. This helps protect from attacks as listening to network traffic would not help attackers narrow down anything to gain useful information. After sending the challenge, the server goes to the next state to wait for the client’s response. The client then encrypts and returns the message. The client goes to waiting for a verification to continue from the server. The server first decrypts the response to the challenge. If it still matches what was sent, then the server sends a verification back and waits for the client’s challenge. If it was not correct, then the server terminates the connection. The client creates a challenge in the same way that the server did and then sends that and waits for a response. The server also encrypts the challenge and sends it back. The client attempts to decrypt and as long as they get the expected message they go to the next state, which is sending their SID. The server went to waiting for a SID from the client after sending its response to the challenge. The code then carries on with the rest of the information sharing. One of the primary attacks that had to be guarded against with this authentication models was a reflection attack. In the reflection attack, there is a change in the order of packets sent allowing for a clever attacker to get the information and fool the server. First, the attacker starts a session and sends his information and a challenge. In this model, the attacker has an advantage because they send the challenge first, making the server “prove” itself to them first. The server then responds with the encryption and its own challenge. The attacker can then open a second session and then send the server’s challenge from the first session as its challenge. The server would respond with the correct encryption which the attacker could then use back in the first session to bypass security. My implementation avoids this because the client that is connecting has to verify itself first before the server will ever send anything encrypted. By the time the connector has the information of an encryption, they have already been verified.

# Deduplication Solution:

Another problem that the system has is that it does not appropriately deal with duplication. This section will discuss how this was handled in the implementation following constraints. Other options that would be better designs will be discussed in another section. The initial problem was that the drone broadcasts its message to all the towers that are within a certain radius. These towers then have to replicate their data to the other servers to make sure that all the servers are up to date on what is going on and can track the drone. The problem is that there is no check and since the towers have different timestamps, there is no guarantee that the timestamps will match up. If they do not, they will mess up the drone’s flight path which would make it harder to calculate. In order to address that I use an algorithm that eliminates duplicates. It runs in O(n3) time with the worst case being all replicated data that is unique. Each time a piece of data is replicated in, the server compares it against every point that is in its current database. If there are no duplicates then it keeps it. If there are duplicates, then it picks whichever one was entered second and gets rid of it. It then scans the rest of the database to ensure no other duplicates happened. Comparing each plot against each other plot is an O(n2) task. This would be done at most n times if all the data was replicated. It has to be done this way because data picked up by the antenna that is paired with the server was not code that we were supposed to edit. Since we are unable to edit that code, we have to check the whole database to catch any duplications from an antenna injection that might have happened after a replication for that data already happened. One more check is also done at the end just to ensure that the data is correct. One such sweep would be sufficient if the data only had to match at the end. However, the data was supposed to be consistent so that duplicates were taken out as quickly as possible so waiting until the end to do so was not possible in this scenario. The code checked the plots so that it could handle different drones being in the same place and not remove the data. They also handled time differences. If the drone flew back along the path that it flew out on, as long as there is a 20 second difference between the coordinates matching, the code can differentiate between the two. I noticed this problem in my testing and found a way to fix it by adding an additional check. I selected 20 seconds because the system documentation states that drone timestamps are with 15 seconds of the global clock. Additionally, the severs are with +/- 3 seconds. Therefore, that means a max of 18 seconds difference, although the servers can only be up to 6 seconds different. I felt that this amount would be able to tolerate an acceptable amount of flexibility and could only potentially lose a couple points in a scenario where the drone is trying to immediately turn and return on a straight line.

All in all, this part (and the time skew) probably took me 35+ hours to complete (not including the authentication). I didn’t want to turn in a project that had not completed this part so I decided to keep working and finish it instead of turning in an incomplete project.

# Time Skew:

The third problem was related to the plots and it is that the servers are not synced on time but the server uses its time to timestamp each piece of information it receives. This leads to the drone seeming to speed up or slow down when the server’s points do not match. There are two steps to fixing this problem. The first is having a leader so that all the servers know which clock to base their final results around. Finding a leader is done through an election. In my code the election consists of getting all of the server’s names and then picking the one with the lowest ID. The instructions stated that this election should not send packets for coordination so it does not. The next step is correcting the time differences on each server. First, any time there is a duplicate that includes the local server, I calculate the difference between the local server timestamp and the other server’s timestamp. I store that information and use it with future points and to fix past points. I have to use the local time until the end because we are not supposed to touch the antenna simulation code so I can’t modify the injected points which means when I adjust them at the end, they all need to be based on the local time. At the end of the code before printing out, I run through the list again and reverse the difference between the local server and the leader and that puts the file in terms of the leader instead of local.

A basic pseudocode algorithm where if an offset is known prior to entering it in the plot, the offset is applied

If plot1 and plot2 are duplicate and don’t have all offsets for both:

Check if one of them is from the local server

If so get the offset with foreign.timestamp - local.timestamp = offset

Apply said offset to all previous entries with foreign.SID

Else If neither of them is local but one of them is already corrected to local time

get the offset with corrected.timestamp - unknown.timestamp = offset

Apply said offset to all previous entries with unknown.SID

My code has this working however, it can still have occasional inaccuracies due to data precision problems and potentially some adding time challenges. Within the output it is synced properly so it displays every 5 seconds. (Example of output is shown in PROOFsvrX.txt files).

# Election:

The assumptions for a coordinator include that all processes are the same which all servers are the same. The next is they all have a unique ID and they all know all the other ID’s. Processes don’t know if the other servers are online. In the system, one server might not be run, and the system just continues sending to data to them when they get it. My implementation selects the process with the lowest ID. In the current setup that will be ds1. The project said that sending packets for an election process is not required, so each server looks at the list of IDs and then decides that the lowest one is the leader by itself. If packet sending was involved it would likely evolve to a bully algorithm. In this algorithm a process holds an election and sends packets to other nodes until a higher one responds (if searching for the highest node to make it a leader). The higher node then starts its own election until the highest is found. It then sends a message to the rest saying it is the leader. In the implementation, each server assumes that every server is up and then picks the lowest one. Actual implementations of these elections would probably consider something more important such as connection strength or power ability or shortest overall distance to other servers.

# Coordination techniques:

The problem of dealing with replicated data and clocks not beginning completely synced is one that almost all non-trivial distributed systems have to account for. The way this system dealt with these problems is not the most efficient nor does it follow a standard way of doing things. In this implementation, choosing to sync the clocks was not a possibility but there are some current solutions that do so and are helpful to solving the problems the challenge presents.

One possible solution to the problem would be to have the drone timestamp all the packets it sends out. Then instead of worrying about what time it is on each node, the time will come with the information meaning that the timestamps are all already synced. Additionally, deletion is easy because if there is an entry from the same drone ID at the same time, it is a duplicate. Delays are not as important from a consistency standpoint because the time will not change based on those delays and the drone will always be accurate to itself. A potential drawback would be if someone needed to understand the ordering of when a drone flew to one place versus when another drone flew to the same place, if the clocks are synced, the wrong conclusion is possible.

Another potential solution is the Network Time Protocol. This protocol has the servers send messages to each other with timestamps and estimates the delay between the two. If the serer that is changing its time can get more accurate with the information provided it does so. This algorithm slowly corrects over time. While this would not be helpful in a situation where the antenna starts up as soon as a drone starts flying, in the real world, the antennas would likely always be on so this would allow for times that would be sufficiently close enough. This algorithm has an accuracy of 1-50 msec worldwide. Since the antennas are only covering one city and the timestamps are in seconds that is more than sufficient for our purposes.

Vector Clocks also seem to be an idea that could apply to this scenario. While Lamport clocks can tell if an event *a* happened before event *b* it cannot tell that based on the time values of C(*a*) and C(*b*). Since this system stores the time points, being able to be sure that the times mean that the event happened in that order is essential. A vector clock captures causality which connects to the consistency model discussed below.

# Consistency Model:

Consistency models are agreements between a data store and the processes that use that data store. If the process follows rules, the data store will remain consistent in how the data is stored there. Data-centric consistency models aim at providing a systemwide consistent view on a data store. It assumes that the data is available immediately via replication. On the other hand, client-centric consistency provides guarantees for a single client concerning the consistency of accesses to a data store by that client. No guarantees are given concerning concurrent accesses by different clients.

This model seems to be more like a data centric consistency model. The goal of this system, properly designed, was that a user could look at any of the servers at any given time and know where the drone was and how fast it was going. That would make the ideal consistency sequentially consistent. However, due to the need for deduplication and changing time stamps and the fact that sometimes you will have information that came from one server and on the other server that entry came from a different server based on how you deal with duplication. That being said, I think that the data-centric casually consistent model is the closest to this implementation. Data is discovered and then it is sent out to everyone else, even if they aren’t asking for data. They then deal with the data and clean up duplications.

# Communications method:

As stated, lower-level communications are handled in the TCPConn class. It creates tags so that the type of message is stated around the data. There are functions to handle wrapping the commands and sending them. The authentication that I implemented uses these tags to establish they are authentication messages. There are also already functions to send and receive data. This system uses a Message-Oriented communication where the information is sent over sockets and follows a TCP implementation. In this system, the communications are not persistent. Each time there is a communication to be made, the two sides connect, do authentication and pass their information and then disconnect. This makes the idea of having a leader less effective because it doesn’t make sense to consider a leader going down since if that node went down it just means that those communications don’t happen. This is because the system does Multicast where it sends all the information that needs to be sent to everyone and doesn’t check for confirmations. In this case the information is Drone Plots that need to be replicated. This works because it is a peer-to-peer system. Since there are few nodes, any server can easily send it to other servers. As the system expands, instead of a graph approach perhaps a tree or another topology would become a better implementation.

# Naming convention:

The naming conventions used in this project are a SID which is a string that the server is called. It is assigned that name by comparing against what IP and port number the server is running off of. The system maintains a name-to-address binding which allows the system to know what the human-friendly name refers to. The idea in this implementation seems to try to follow structured naming principles in that the name is still composed from a simple, human-readable name, but that name is attached to useful information for the computer. The names for the server are stored on a file that all the servers have access to and they read it in so that is how the information is initially distributed.

# Security:

Although the finished system now has more security for connections through the authentication process, it would not be considered secure. The database holding the records is not protected so that would need to be secured. The shared key is on the all of the systems but if it is lost or stolen there is no way to establish and share a new key (at least within the system we were shown). The information is not locked down as well so if someone wanted to attack our drones or figure out where things were being sent, they could likely steal the information to do so. Essentially there is some hardening of the system, but there is still a significant amount of security that needs to be implemented. Since there is not back-and-forth coordination between nodes, if one was compromised and started sending bad data, that could completely compromise this set-up.

# Other concepts:

The type of distributed system is a distributed information system. This system, or at least the functionality we can see,

One important distributed systems concept is the idea of transparency. The first type of transparency is access where the differences in how data is stored is hidden from the end user. This system excels at this because assuming the end user only gets the output from the end, they have no idea about the transformations or anything else done with the data throughout the process. They only have the final product. Location transparency is next. The location of the antennas is unimportant as long as they all (combined) cover the entire location for the drone. Relocation transparency is somewhat there. The antennas don’t seem to completely overlap but there could be some relocation that would go unnoticed. Another type of transparency is replication transparency. If this system were completely created correctly, the end user would have no idea that data was replicated unless the carefully compared all the logs and saw some points from the same drone at the same location were caught by different nodes. Still the replication is transparent because all duplicates are removed and the time skew should be corrected. Failure transparency is one concept that I really don’t think this system has. In order to have it, all points would have to have a duplicate, which is not the case in the scenarios we were given to run in this project. Otherwise if one antenna went down and it was the only one that could receive the drone’s information, that information would be lost.

Notes:

* I included everything in my repository because it has some additional proof documents and it has the make files that I used. If you use other files make sure that Deduplicate.cpp is part of the make file as that handles duplicates and time skew for me.
* Location of repository: <https://github.com/chrisvoltz19/AFIT-CSCE689-HW4-S>

Assumption:

* the node name must be “dsX” where 'X' is the SID which is an unsigned int