

Summary: I am working on drone swarm communication. The goal of the project is to enable file transfers within and out of a simulated swarm by implementing network functionality from the ground up using OpenFlow and application-level logic. The project steps (numbered below) have not changed. In each section, I detail what work has been completed or needs to be done, along with my thoughts on how I will approach it.

1. Set up simulation
2. Implement a neighbor detection algorithm
3. Implement a path selection algorithm
4. Implement data exchange within the swarm
5. Implement data exchange external to the swarm
6. Scalability improvements

Set up Simulation

I am using Mininet to simulate a wireless network. The OpenFlow controller platform that I'm using is POX. I've installed a VM with Mininet and its associated tools and have written a basic Python script capturing a single network topology and associated controller. There are a couple of advantages and disadvantages to using Mininet.

The advantages are that I can easily define a network topology, change the topology, take advantage of processes running actual TCP/IP stacks, use OpenFlow to program switching behavior and that hosts can run arbitrary applications.

The disadvantages are that I can only run OVS/Openflow on simulated switches (not hosts), POX supports only OpenFlow version 1.0 and therefore has limited programmability, and only ethernet links can be directly simulated.

I think that Mininet will still be adequate for simulation purposes, however. I can specify the behavior of individual links (ie. latency, packet loss rate) to roughly simulate wireless transmission. However, since only switches can run OpenFlow, not hosts, I need to get creative with how the network is represented and how the protocols are coded so that they are functional in the context of the simulation itself. For now, I have connected all hosts to a switch which has OpenFlow rules pushed specifying which hosts a packet from a given host can actually reach. That is done as part of the configuration script. When the time comes, I can simulate network topology changes by changing flow rules. Call this topology 1.

There are a lot of downsides to topology 1. While my code has thus far been written for that, I would like to move over to a topology where every host has its own OpenFlow switch and the switches connect to each other to simulate a connected topology. Call this topology 2. This would allow me to push some of the neighbor detection and routing logic to the switches, rather than the application level as described in later sections. It would also make the simulation more realistic. There's a good chance that I'll switch over to that topology before the next checkpoint.

I considered using Mininet-Wifi instead (it adds wireless links and access points to the base model) and, indeed, I could easily get an adhoc mesh network running with it just using the base simulation

software. But modifying routing in adhoc mode with Mininet-Wifi would be very complicated (I believe I'd have to write it in C++ and somehow integrate it in). Instead, I could use a similar topology as above by swapping the switch for an access point. Based on what I've read, it should be straightforward to switch over. However, then all hosts would interfere with each other rather than being grouped properly and only communicating with neighbors. Another alternative would be for each host to be connected to its own AP and do AP to AP communication with mobile APs. The problem is that solution requires hardware support that my laptop doesn't have. So, for the purposes of this project, I think it is more feasible to represent the network using ethernet.

Below, I show implementation plans for topology 1 and topology 2. I will only do one of them, and it will likely be topology 2. The code in the repo is for topology 2.

Implement neighbor detection (topology 1)

I am planning to implement neighbor detection using a combination of OpenFlow rules and application-level logic. The switch needs to statically know the entire network topology at the start of the simulation. For this, it needs rules for dropping host-host packets that aren't allowed by the network topology and it needs rules for replicating host-host packets to all reachable neighbors. Neighbor detection (analogous to ARP requests) will be implemented as an occasional application-level request to a specific IP address/port (irrelevant which, as long as it's consistent). The host application makes a custom raw packet and sends it via the Python socket module. The switch receives the packet, matches on the IP address/port, and forwards to the reachable hosts. These receive the packet and send back a response packet. At the application level, hosts would maintain a map of reachable IP addresses. This packet might be sent out, say 10 times, to deal with packet drops and generate link quality information. To deal with topology changes, if a request goes out to a known host and no response is received for all requests, that host is removed from the list.

Implement neighbor detection (topology 2)

If I swap over to a topology where each host has its own switch, then the rules can be all pushed down to the switches/controllers and they don't need to know the entire topology beforehand. Each switch would only need to be programmed initially with a flow rule specifying that their connected host is on a specific port and to have that information available to share. I will now describe the software running on any given switch. It will have a timer which will go off occasionally. When it does, the switch sends a Hello packet on all ports. All receiving switches respond on all ports with: known MAC addresses of hosts, cost to reach them, and whether they have internet access. When a switch gets this response, it compares its own table of this information with the received information, and changes entries based on cost to reach them and whether a host has become known or out of reach, and updates a first hop field with the MAC address of the switch it received the update from. On any changes to this table, a switch sends the list of reachable IP addresses to its host. Switches track cost metrics to each responding switch by counting the number of sent Hello packets vs received responses – emulating a rough ETX metric. This could be improved through a weighting algorithm which puts more emphasis on recent values, but I'm going to implement it is simple counters to start.

Implement path selection (topology 1)

Path selection will be done primarily at the application level. This is because 1) the proposed simulation topology isn't amenable to doing it at the switch level and 2) data exchange will be done at the application level to ensure data is kept safe if the topology changes and no path to the destination is available.

Network mapping: each host application will have a list of all the other hosts it knows about, a cost to get there based on ETX, the IP address of the first hop to get there, and whether that drone has internet access. Similar to above, a special packet will be sent to all neighbors and a response will be expected. This packet may be sent, say 10x a second, and an ETX value calculated based on the number of received responses. Since links cannot be simulated to have a different drop rate depending on direction, the ETX value will be for a round trip.

Sharing routes: each host application will occasionally share its list of known drones, costs, and first hops with its neighbors. On receipt of a list of known hosts, a host will compare that list to its own. It will add entries that aren't present and it will replace entries for which a lower ETX is available (it adds the ETX to the next hop to the value it received). If a host's list gets updated, it will send its updated list to its neighbors. If a host has a destination with a given first hop, and it receives a list from the first hop that no longer has the destination in it, then it removes the entry from its list.

Implement path selection (topology 2)

If I swap over to the topology where each host has its own switch, path selection is easier because the cost metric and list of reachable hosts/first hops is generated by hello messages. The data is simply forwarded to the next hop. If the recipient is unreachable, the controller will store the packets for a short time before deleting them. If the destination becomes available before then, the packets get forwarded. This can be improved by adding versioning to packets, but I will leave that for future work.

Implement data exchange within the swarm (topology 1)

Drones must be able to store another drone's sent data if the topology changes and a path to the destination drone is no longer available. I propose that a host application on a specific port should be dedicated to data exchange within the swarm so that data can be stored in user space until a path becomes available. A drone would send its data to the next hop's data handling application via FTP, along with the final destination IP address. The data handling application would then attempt to forward the data. In the event of a failure to send due to timeout, the application would occasionally try again to send the data. If the drone is the final destination, it keeps the data. For data that is being held until a path becomes available, a timer should be in place to delete it if it becomes too old. Similarly, sent data should be associated with a version number. If a drone holding data receives a newer version of that data, the old version is replaced.

Implement data exchange within the swarm (topology 2)

Packets sent to an IP address within the swarm are forwarded, stored, or deleted as described in the path selection section. Packets should be sent via TCP so that the application knows to resend the packets if a response is not received. I know that this will decrease performance when the number of hops increases, but I want to start simply.

Implement data exchange external to the swarm (topology 1)

I propose that drones with internet access advertise it as part of path selection. Then, similar to data exchange within the swarm, data will be sent to a specific port via FTP which is responsible for holding onto the data in case of the path to the destination becomes unavailable. The data is sent to the next hop, along with the destination IP address. If the drone has internet access, it sends it to the internet using FTP.

Implement data exchange external to the swarm (topology 2)

Similar to data exchange within the swarm, but if the packet is destined for port 80, then it is routed to the host with the lowest path cost that has internet access.

Scalability improvements

The path selection, congestion awareness, and data exchange can all be significantly improved over what I've suggested. Once the system is working, I will solicit feedback on what I might focus on. I will also stress test it (ie. change the number of drones in the swarm, change the time between network changes, change the neighbor detection frequency, etc.) to try to identify failure points.

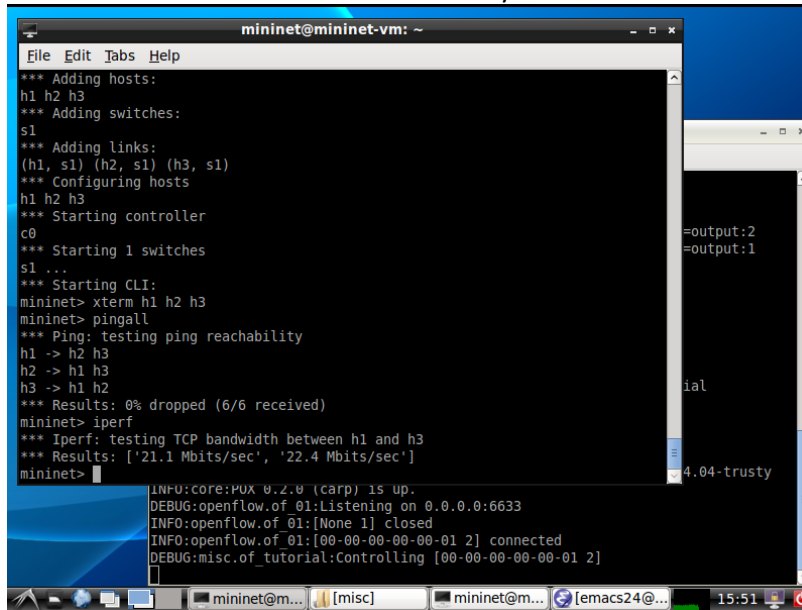
Timeline

I have the basic simulation set up now. Next is swapping over to topology 2 and implementation of neighbor identification and path selection. Data exchange will then be fairly straightforward application-level logic. My goal is to have implementation done by the next checkpoint. Then, I can focus on testing the system, writing the final report, and making some improvements to the system for the final checkpoint.

Screenshots for check-ins

I haven't pushed any code for the check-ins yet but I do have Mininet working and a basic single switch topology communicating properly, along with a nice python configuration script describing the topology and a basic script for the OpenFlow controller. I've spent a lot of my time thinking about the tools available to me, their capabilities/limitations and the architecture required as a result. I think that spending my time thinking about logistics was better than jumping in head-first. I'm going to start (and hopefully finish) coding during this iteration. I want to get most of the code written this weekend. The code pushed as part of this checkpoint is the code that I've written so far in support of the topology 2 rewrite.

Here's a screenshot to show that I actually have a basic Mininet configuration working 😊



```
mininet@mininet-vm: ~  
File Edit Tabs Help  
*** Adding hosts:  
h1 h2 h3  
*** Adding switches:  
s1  
*** Adding links:  
(h1, s1) (h2, s1) (h3, s1)  
*** Configuring hosts  
h1 h2 h3  
*** Starting controller  
c0  
*** Starting 1 switches  
s1 ...  
*** Starting CLI:  
mininet> xterm h1 h2 h3  
mininet> pingall  
*** Ping: testing ping reachability  
h1 -> h2 h3  
h2 -> h1 h3  
h3 -> h1 h2  
*** Results: 0% dropped (6/6 received)  
mininet> iperf  
*** Iperf: testing TCP bandwidth between h1 and h3  
*** Results: ['21.1 Mbits/sec', '22.4 Mbits/sec']  
mininet>   
INFO:core:POX 0.2.0 (carp) is up.  
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633  
INFO:openflow.of_01:[None 1] closed  
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected  
DEBUG:misc.of_tutorial:Controlling [00-00-00-00-00-01 2]
```

How the project will be evaluated/validated

There are a few interesting ways this system can be evaluated. One is to determine how long it takes the system to settle/identify the optimal routes after a major topology change. Another is to determine

how long a file transfer within the swarm takes. Some factors which could be varied include swarm size and packet drop rates.

Summary: I propose to work on drone swarm communication. At a high level, the project can be described with the following series of steps. Each step is described in greater detail below.

7. Set up simulation
8. Implement a neighbor detection algorithm
9. Implement a path selection algorithm
10. Implement data exchange within the swarm
11. Implement data exchange external to the swarm
12. Scalability improvements

Set up Simulation

One aspect of the simulation involves specifying which drones can communicate with each other. This may be specified as a list of drone pairs, where each pair can communicate. It can be modelled in a simulation software as a set of trees of routers and hosts. Alternatively, it could be modelled in Linux such that all drones are modelled as hosts all connected to a router whose forwarding behavior is controlled by iptables.

Another aspect of the simulation is the time element. The network topology should change over time, so a clock is required to determine when the topology changes. The network can transition between multiple topologies simply by using multiple lists of drone pairs and changing from one list to another after a certain amount of time has passed.

Implement neighbor detection

I propose that drones ping hello messages every x seconds with a sequence of Hello, Ack, Ack. Each drone must also be capable of running BGP to identify whether it can connect to the internet. Separate routing tables should be used for destinations within the swarm and destinations exterior to the swarm.

Implement path selection

I'm not sure how I want to do this yet. Congestion-awareness will be very important, but I haven't decided whether I want drones to make locally optimal decisions (ie. a BGP-like protocol with congestion awareness a la DCTCP) or globally optimal decisions (ie. full swarm awareness with congestion awareness based on RTT and/or available bandwidth). I may start with the locally optimal approach. To deal with changing topology, it will be necessary to either store multiple paths to a destination or to run neighbor detection and path selection again when multiple timeouts are encountered.

Implement data exchange within the swarm

Drones must be able to store another drone's sent data if the topology changes and a path to the destination drone is no longer available. I propose that an application should be dedicated to data exchange within the swarm so that data can be stored in user space until a path becomes available. A drone would send its data to the next hop's data handling application via FTP, along with the final destination IP address and socket. The data handling application would then attempt to forward the data. In the event of a failure to send due to timeout, the application would occasionally try again to send the data. If the drone is the final destination, the application will forward the data to the correct

socket. The performance can be improved through use of DPDK, XDP, or PF_RING. I propose to initially use the default TCP/IP stack.

Implement data exchange external to the swarm

I propose that drones with internet access advertise it as part of path selection. Then, similar to data exchange within the swarm, data will be sent to a specific application running on each host via FTP which is responsible for holding onto the data in case of the path to the destination becomes unavailable. The data is sent to the next hop, along with the destination IP address and a flag indicating it is external to the swarm. If the drone has internet access, it sends it to the internet using FTP.

Scalability improvements

The path selection, congestion awareness, and data exchange can all be significantly improved over what I've suggested. Once the system is working, I will solicit feedback on what I might focus on. I will also stress test it (ie. change the number of drones in the swarm, change the time between network changes, change the neighbor detection frequency, etc.) to try to identify failure points.