CS5590SP: INTERNET OF THINGS

# Mobility Inspired SofTware Defined Networking -Software Defined Mobility

## Introduction

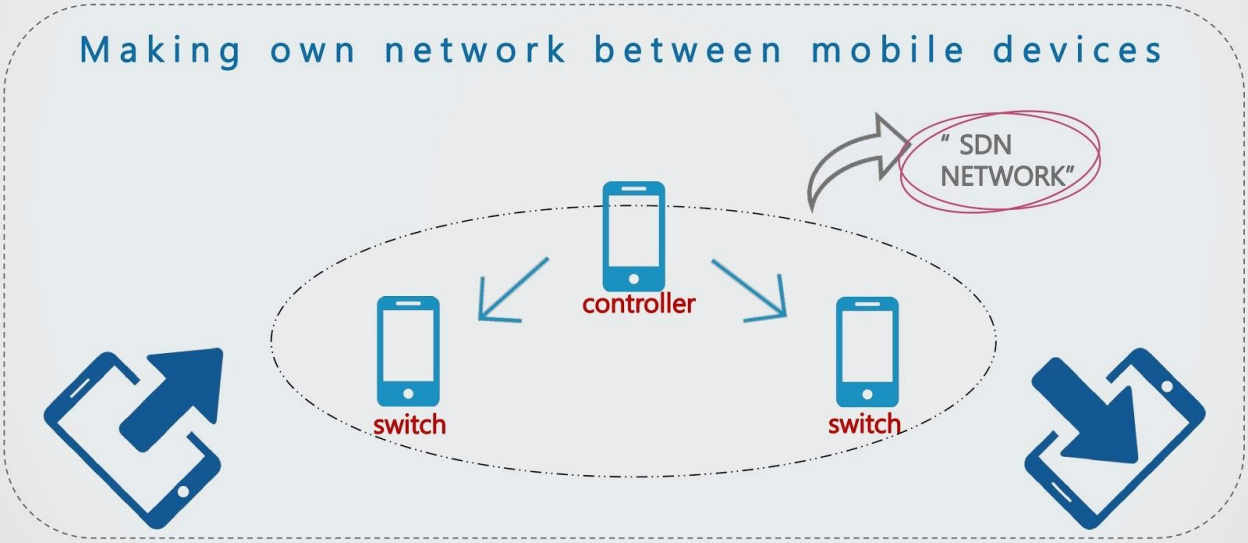
Traditional SDN has been applied to solve growing complexity issues in networks, but how can it work in a more mobile environment? This inspired us to develop Software Defined Mobility.  Traditional SDN has two major parts 1.Controllers and 2. Switches (Forwarding devices) These controllers are static. In an ever-changing environment, even the controllers need to be dynamic.

Now, we propose to make the controllers themselves mobile. A truly mobile network can add or subtract nodes, even controllers and continue to function. In order to have some

control over the ad-hoc nature of mobile computing we propose - MIST (Mobile Inspired Software Defined Networking). MIST has a mobile adhoc network using smartphones devices that will act as a both controller and switch. The reason why we need software defined mobility is because traditional network is static. However in mobile environment, network nodes are easily changed. For example, mobile battery out, move away from network, also some kinds of error can happen. Because of these reasons, dynamic network needs.

## Goal:

fig.1 Goal of our project




Our primary goal of the project is to have a smartphone that can act as both controller and switch. Then we wanted to implement dynamism meaning that a device can join or leave the network without breaking any topology. After this we also wanted to implement synchronisation between the controllers. And future would be forwarding request to the controller when there is a missing in flow table at a switch, updating flow tables, master controller can write some configuration to the forwarding devices and various other algorithms to make the system robust.

## Implementation:

We implemented the following as an android application.

1. Implement mobile version of Libfluid
2. Connection establishment and data transfer between controller and switch
3. Dynamic controller election when a controller is died.

Following Fig 2. is the topology of our system. Each Android device can act as a controller, switch and host. Host is the one that initiates the request of packet transfer from one host to another host. Then the request reaches a switch which will determine the next forwarding device from looking at the flow table. If there is match it will forward the packet otherwise it will ask the active controller for the path. There will be only one active controller at any point of time in the network. All the active forwarding devices(Switches) connects to the active controller. During this time controller part of the forwarding devices will be in standby mode. Each host connects to at most one switch to be identified on the network.

Fig 2. Topology of the system

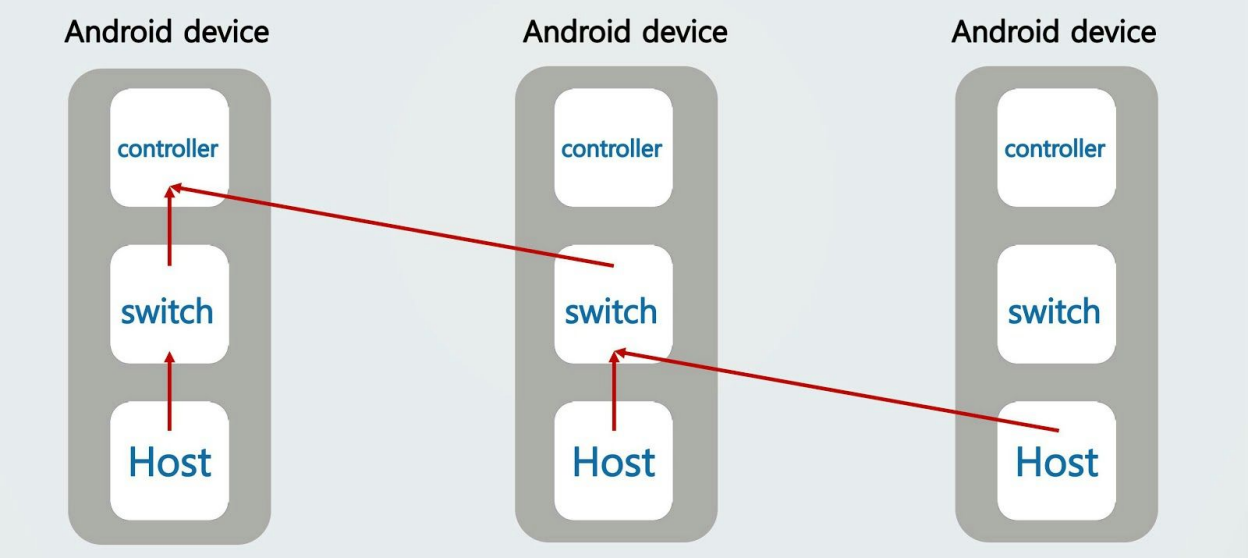


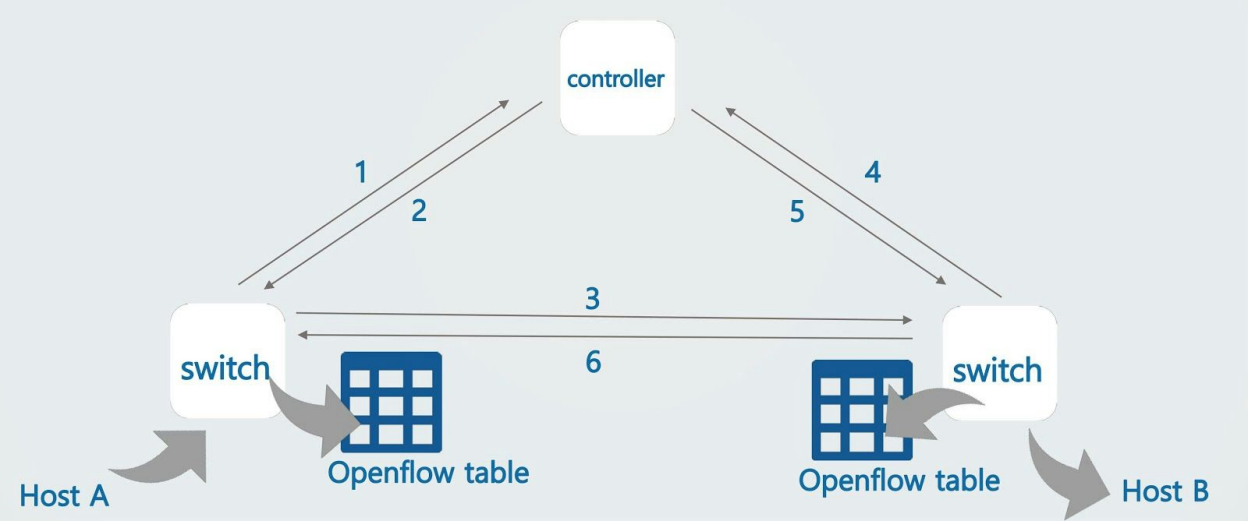

Fig.3 depicts the message flow control in the systems

1. Host A sends a packet to destination Host B but the information of the destination is missing in the flow table. So the Switch asks the active controller for the missing entry in the table.
2. Controller replies back the entry to be updated in the flow table.

3. Switch updated its flow table and forwards the packet to the next available destination and from there to destination.
4. Now Host B wants to reply some acknowledgement to Host A. Since the entry at the switch is missing from Host B to Host A, it will ask the controller to give the entry.
5. Active Controller replied back with the entry.
6. Switch will then forward the packet to destination switch and from there to Host A.

From now on Host A to Host B communication is already determined they don't have to talk to the controller anymore until there is a miss in the flow table.

Fig.3 Flow of operations




## Challenges:

Following were the challenges faced by us during project implementation. Fortunately we were able to determine workarounds/ solutions to these.

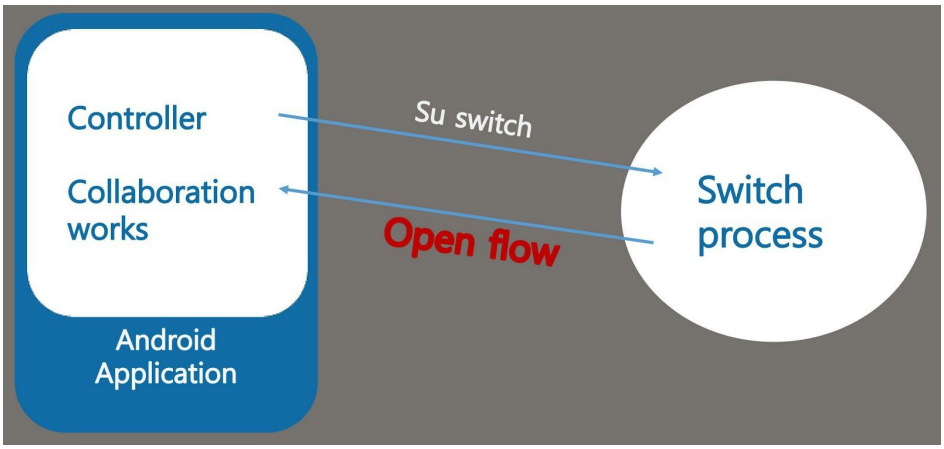
1. Cross compiling issues when libfluid is ported to Android application -- We built a separate tool which is included in the project repository no special care has to be taken now.
2. Root privilege issues to capture packets -- Discussed more in the software architecture section.

3. App permission is not enough to run the application as a superuser -- Discussed more in the software architecture section.

## Software Architecture:

One the main problems we faced with this project was root privilege/ superuser privilege of an application. Android does not allow an application to run as a root application instead it will run with user privileges. Why do we need root privileges in the first case. Since the switch should be able to capture the packet from the interface we need root privileges. Moreover, each controller should be able to scan devices (let's call this arp scan for simplicity) on the network to determine who is the master. To resolve this issue our architecture is the solution. Instead of running the switch or arpscan as a seperate thread we built them as a seperate executable files (build script can be found in Android.mk file). Then we will start them as separate process from android application. Whenever we require them we start them as a separate process. This enables them to run at process level giving root privilege. Controller is divided into two parts. Core part is implemented in c++ (libfluid). The synchronization part between all the controllers is implemented as an Android application.

Fig. 4 Architecture of android application




## Conclusion:

Finally, we successfully can show the demo for the goals we specified in the beginning of the project. This project sets the foundation for a much bigger picture of Software Defined Mobility. The demo scenario can be seen in the Fig.5 below.

Fig.5 Demo scenario