Decentralized Device-to-Device Chaining (D3C) Protocol

University of California, Santa Barbara
Jared Fitton - Garrett Lee

1. Abstract

The basis of this project is developing an alternative form of communication for mobile devices which can be used as a solution for two problems. The first problem this would help solve is reducing the stress on existing WiFi and cellular infrastructure as it provides another means of sending data. The second problem is providing a viable means of connection between devices given that standard technologies such as cellular go out of service or are unavailable. While there may be other means of communication available in certain areas, such as WiFi, this may not be an option for everyone and as such a different method must be used to make contact with other devices. In this project, devices will use their personal area networks (PAN) to be able to determine what other devices are within their range. On identifying other devices, a device will establish connections with them to exchange information about the connections they have, so as to form a valid network of devices. Each device will then subsequently act as a switch within the network routing messages between other devices while also being a valid destination to receive messages itself. This project focuses on creating and implementing the algorithms that would be used to create a network of devices. In addition, it covers how to exchange routes and route messages between devices.

2. Project Invariant

One of the biggest problems that we faced while trying to devise a valid routing protocol was determining how to establish the initial connections between devices in addition to figuring out who to advertise to. An important factor that we would have to consider in our network would constantly move in and out of range of numerous devices which is much more dynamic than the static networks we are used to. To reduce the amount of overhead within the network, we want to have the lowest amount of devices that act as both hosts and peers since they will generate double the amount of traffic if they were to have only one of the roles. Ideally, we could create a network that could accompany any number of devices regardless of how the devices are situated relative to each other. However, to create a protocol that would be able to do this would be very complex as we would have to consider a multitude of edge cases. We originally tried to take on this complex problem and create such a network, but we quickly ran into many roadblocks. Since we had to create an algorithm that would have to automatically set up the connections between the devices in opposition to already having the connections made, the complexity would be greater than that of standard routing. This complexity arises from the fact that we need to have our algorithm choose which device will host when there are multiple devices within the same vicinity of each other. In this case, each device would have to determine by itself whether it was a peer or a host and we would have to have every peer only connect to the host otherwise there would be unnecessary connections which could lead to routing loops later on. This is especially difficult for devices that are trying to be added to the

edge of the network as we only want one connection to be established so that no routing loops occur. To add to this, we would also have to deal with devices leaving the network which completely changes the available routes from each device and as such the routing tables of every device on the network would have to be fixed to account for the device leaving. This would be a very uncommon scenario in today's current Internet architecture as routes are generally static and do not change very often.

To be able to make progress on our project, without being held back by not having a complete protocol, we decided to create an invariant that when upheld would allow us to simplify the design of our algorithms. The invariant for our protocol is that devices will only be added linearly to the edge of the network. To imagine adding linearly, you can envision devices located on a one dimensional line, and when a new device gets added to the network they can only be connected to the furthest most device at the left or right of the line. While we understand that this will limit the overall usage and application of our protocol, we are just using it as a base board to create an implementation that will work in a three dimensional space. It also exists to illustrate that our project idea works as a proof of concept and can be improved upon in the future.

3. MultiPeerConnectivity Framework

We wanted to make our proof of concept run on iOS devices, therefore we used Swift to program our project. From the available libraries that we could use to create a PAN we decided on the MultiPeerConnectivity Framework as this allowed us to easily discover peers and establish connections to other devices. The MultiPeerConnectivity Framework architecture allows each device within a peering relationship to have either the assigned position of host or peer. A session is established by devices that have the role of host, inviting other non-host devices, otherwise known as peers, to their session. Within each session there is only one host but there can be multiple peers. Every device is capable of establishing multiple sessions at the same time and thus can act as both a host and a peer simultaneously. The MultiPeerConnectivity Framework is composed of Session, Advertiser, and Browser objects which are used to create and maintain the sessions among devices. The session objects are used to maintain a session between a peer. Peers will be added to the session when they accept an invitation and sessions will be created on devices that are acting as peers and accept the invitation. Advertiser objects are used to advertise to other devices that the current device is looking to join a session and will be used to accept invitations from other devices. The Browser object is used to search for other nearby devices that are looking to join a session and from this sends invitations to those devices so that they can join this session. Every device is given a unique PeerID that is used to identify itself.

4. D3C- Protocol

Two Initial Devices:

1. When our app is launched, the device will start out with both the advertising and browsing handlers running. Every device will begin with an empty list for each device that it is connected to, and this will contain all of the routes that can be reached by routing traffic through that specific device.
2. When two devices come into range of each other, they will look at each other's peerID within the advertisement and browsing packets that have been sent by each device, and the device that has the higher peerID will be the one that will assume the role of host and the device with the lower peerID will assume the role of peer. The host will then send an invitation to the peer, which the peer will accept and a new session will be established between the two devices. The device with the lower peerID will then initiate a request for routing exchange information during which it will send itself as an accessible device to the other peer. The receiving device (host) will update its routing table with the route names it receives and sends back it's own device name as a routing update to the initial sending device. The initial sending device will also update its routing list with the other device's name. This results in both devices now having the other device in their routing list.

Adding Devices to the Network

3. When a new device comes into range of one of the edge devices, it will be both broadcasting and listening for advertisements just as in step #1. Devices that are able to see this new device will first check their routing lists to determine whether or not a device on the network has a route to the new device so as to reduce redundant connections and prevent routing loops, if a connection does not exist then the device with the lower peerID will then become the peer and the device with the higher peerID will be the host. A session will then be established between the two devices and then route exchange will occur.
4. Similar to the device exchange in #2, the device with the lower peerID will initiate a request for routing information exchange in which it will send the route list associated with the other device that it is connected to, as this will be all of the routes that can reached by going through this device along with the device's own peerID added to it. The receiving device will then do the same and send back the routing list for the other device that it may be connected to. Afterwards it will update the routing list for the newly connected device with the list that it just received. The sending device will then subsequently receive the other device's routing list and update its routing list for that device too. At this point both devices know of each others accessible routes, so they each send out a route broadcast update which updates the routing tables of all the other devices on the network, except in the direction of the device we connected to. Now every device has a full view of the network and should know where to send a message for it to reach its destination.

Sending Messages

5. Every device on the network now has knowledge of all the available devices on the network that can be reached and which device they should send traffic through to get the data to its destination. To send messages and properly display them we need to have a message format that includes the destination device along with the sending device so that the receiving device will be able to tell who the message is from.
6. When a device decides to send a message it will look at its routing list for the devices it is connected to, and then will determine which list the destination device is in and from that will be able to determine what device should be the “next hop” in the routing. It will then send the message to that device. Whenever a device receives a message it will first check the destination device and check if that destination device is itself, and if it is then the process is done and the message will be moved up to the application layer. If the destination is something other than itself then the device will look at its routing table and determine which device it should route the message through. This process will repeat continuously until it reaches the destination device.

Sending Acknowledgement Messages

7. Once a message gets to its final destination it will send an acknowledgement message back to the original sender. This allows us to do two things: confirm that the message was received and check the round trip time of the transmission.
8. When a device goes to send an acknowledgement message, it is essentially treated the same as a standard message but in reverse. The main difference is that we include a flag to let the receiving device know it is acknowledging the receival of the original message, the body of the message is the time it was received, and the sender and destination fields are switched. The routing is treated the same as a regular message.
9. Once the acknowledgement message is received, the original sender can take the time from the body of the message and calculate the round trip time by subtracting the time the message was initially sent. This also confirms to the devices that the message was received.

Devices will constantly be exchanging two types of data, the first being routing information and the other being actual messages between devices. We therefore need a way to be able to determine how to distinguish the content that is being received. Therefore, we decided on incorporating some sort of header or flags that would indicate the type of message that is being sent so the device would be able to know what to do with it.

5. Implementation

Now that we have seen a high level overview of how different types of messages are sent across our network, let's take a look into how we actually implemented it.

```
120     public func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {
121
122         guard let message = try? JSONDecoder().decode(Message.self, from: data) else {
123             print("Error decoding message from: \(peerID.displayName)")
124             return
125         }
126
127         switch message.flag {
128         case 0: //Standard Message
129             handleStandardMessage(message: message)
130         case 1: // Routing Info Update
131             handleRoutingInfoUpdate(message: message)
132
133             // Device has been updated with routes and can connect to other devices now
134             if MPCManager.instance.deviceIsConnecting {
135                MPCManager.instance.deviceIsConnecting = false
136             }
137
138         case 2: // Routing Info Request
139             routingInfoRequestResponse(message: message)
140
141             // Device has been updated with routes and can connect to other devices now
142             if MPCManager.instance.deviceIsConnecting {
143                MPCManager.instance.deviceIsConnecting = false
144             }
145
146         case 3: // Ack Message
147             handleAckMessage(message: message)
148
149
150         default:
151             print("Unknown flag in message")
152             return
153         }
154
155         NotificationCenter.default.post(name: MPCManager.Notifications.deviceDidChangeState, object: self)
156
157     }
158
```

This function belongs to the Device class and gets called everytime a message is sent through the session to a device. At this point we decode the data and then determine how we want to treat the message based off of the flag it contains.

Flags:

- 0 => Standard message
- 1 => Routing Info Update
- 2 => Routing Info Request
- 3 => Acknowledgement Message

Before we look into each of the functions that handle the various flags, it is important to take note of what is happening on lines 134-136 and 142-144. Each device has a field in the MPCManager (The object that stores all the connected devices) called *deviceIsConnecting* that stores whether the device is currently connecting to another device and having its routing tables updated.

Standard message:

```
208     func handleStandardMessage(message: Message) {
209
210         logMessage(message: "Recieved message '\(message.body)' from '\(message.sendingDevice)'")
211
212         let currentDeviceName = MPCManager.instance.localPeerID.displayName
213
214         // Display the message if this device is the message destination
215         if message.destinationDevice == currentDeviceName {
216             // Find the correct RouteUI object and update the last message
217             for route in MPCManager.instance.routeMessages {
218                 if route.destinationName == message.sendingDevice {
219                     route.lastMessage = message.body
220                 }
221             }
222
223             NotificationCenter.default.post(name: Device.messageReceivedNotification, object: message, userInfo: ["from": self])
224
225             let time = String(Date.timeIntervalSinceReferenceDate)
226
227             // Send ack message
228             do {
229                 try self.send(text: time, with: 3, senderName: currentDeviceName, destinationName: message.sendingDevice, routingPath: [])
230             } catch {
231                 logMessage(message: error.localizedDescription)
232             }
233
234             return
235         }
236
237         // Forward the message to the next device using the routing table
238         let devices = MPCManager.instance.devices
239         for device in devices {
240             if device != self {
241
242                 if device.routingInfo.contains(message.destinationDevice) {
243                     do {
244                         try device.send(text: message.body,
245                                         with: 0,
246                                         senderName: message.sendingDevice,
247                                         destinationName: message.destinationDevice, routingPath: [])
248                     } catch {
249                         logMessage(message: error.localizedDescription)
250                     }
251                 } else {
252                     logMessage(message: "Destination device '\(message.destinationDevice)' is not in '\(device.name)''s routing table")
253                 }
254             }
255         }
```

The first thing we do is we want to check and see if the message was intended to be sent to this device or if we need to forward it using our routing table. If we determine that this is the correct destination device, then we update our RouteUI objects on lines 217-221. This object is where the UI pulls information from, such as the last message received and RTT, to update the table. On line 223 we send a notification to the front end telling it to update the views with the received data. We now want to send an acknowledgement back to the original sender, which we do on lines 225-235 by inserting the current time into the message body, switching the destination and sender devices, setting the flag to 3 (this represents an acknowledgement message), and sending the message back into the network. The final area left to look at is 237-255 which is where message forwarding takes place. We search the (at most) two connections that the device has and check each of their routing tables, once we find the one that contains a route to the destination device we forward (send) the message to the connected device.

Routing Info Update:

```
259     func handleRoutingInfoUpdate(message: Message) {
260         logMessage(message: "Recieved routing update from \(peerID.displayName)")
261         updateRoutingInfo(routes: message.routingInfo)
262         broadcastRoutingInfoUpdate()
263     }
264
```

There are two steps to handling a routing info update.

1.) Update the routing info for the current device
2.) Broadcast the routing info into the network for connected devices to see

Step 1.

```
292     // Updates the devices routing table with devices it does not yet contain
293     func updateRoutingInfo(routes: Set<String>) {
294
295         logMessage(message: "Update Routing Info...")
296         logMessage(message: "Original Routing Info:\n\(self.routingInfo)")
297
298         for route in routes {
299             if !self.routingInfo.contains(route) && route != MPCManager.instance.localPeerID.displayName {
300                 self.routingInfo.insert(route)
301
302                 //This is where messages from devices will be store, used only by UI
303                 MPCManager.instance.routeMessages.append(RouteUI(destinationName: route))
304             } else {
305                 logMessage(message: "Already contains route to \(route)")
306             }
307         }
308
309         logMessage(message: "New Routing Info:\n\(self.routingInfo)")
310     }
311
```

A lot of this function includes debugging messages that are helpful when you watch the terminal output to see how/when the routing tables update. The real logic is at 298-307. We loop through all the routes in the update message and, if the route does not already exist in the routing table and the route is not for the current device, insert it into the routeMessages array. This array is to determine if the device has a path to a given destination when a messaging is trying to be sent.

Step 2.

```
265     func broadcastRoutingInfoUpdate() {
266         let MPCManagerDevices = MPCManager.instance.devices
267         for device in MPCManagerDevices {
268             if device != self {
269                 do {
270                     logMessage(message: "Sending routing info to next \(device.name)")
271                     try device.send(text: "", with: 1, senderName: "", destinationName: "", routingPath: [])
272                 } catch {
273                     logMessage(message: "Error broadcasting routing info update to
                            \(device.name)=>\n\(error.localizedDescription)")
274                 }
275             }
276         }
277     }
278
```

Now that the routes are updated, we want to broadcast the routing table to the other connected devices in the network. We do so by determining the device that **did not** send the routing info update and then sending our own routing info update to the device in that session.

Routing Info Request:

```
279     func routingInfoRequestResponse(message: Message) {
280         logMessage(message: "Recieved routing info request from \(peerID.displayName)")
281         print("Routing Info: \(message.routingInfo)")
282 
283         do {
284             try self.send(text: "", with: 1, senderName: "", destinationName: "", routingPath: [])
285         } catch {
286             logMessage(message: error.localizedDescription)
287         }
288         updateRoutingInfo(routes: message.routingInfo)
289         broadcastRoutingInfoUpdate()
290     }
291 
```

As you may have noticed, the routing info request is very similar to the routing info update function. The main difference between the two is that a routing info request means that the device needs to send a response message with its own routing tables. It does so before updating its own tables so it doesn't send the information it just received, back to the device that sent it.

Acknowledgement Message:

```
159     func handleAckMessage(message: Message) {
160 
161         if message.destinationDevice != MPCManager.instance.localPeerID.displayName {
162 
163             // Forward the message to the next device using the routing table
164             let devices = MPCManager.instance.devices
165             for device in devices {
166                 if device != self {
167 
168                     // Get the current routing path and add the current device to it
169                     var routePath = message.routingPath
170                     routePath.append(MPCManager.instance.localPeerID.displayName)
171 
172                     if device.routingInfo.contains(message.destinationDevice) {
173                         do {
174                             try device.send(text: message.body,
175                                             with: 3,
176                                             senderName: message.sendingDevice,
177                                             destinationName: message.destinationDevice, routingPath: routePath)
178                             logMessage(message: "Forwarded ack message to \(device.name)")
179                         } catch {
180                             logMessage(message: error.localizedDescription)
181                         }
182                     } else {
183                         logMessage(message: "Destination device '\(message.destinationDevice)' is not in '\(device.name)''s routing table")
184                     }
185                 }
186             }
187 
188             return
189         }
190 
191         let receiveTime = TimeInterval(message.body)
192         let RTT = String(receiveTime!-messageSendTime)
193 
194         // Find the correct RouteUI object and update the last message
195         for route in MPCManager.instance.routeMessages {
196             if route.destinationName == message.sendingDevice {
197                 route.RTT = RTT
198             }
199         }
200 
201         NotificationCenter.default.post(name: Device.messageReceivedNotification, object: message, userInfo: ["from": self])
202     }
203 
```

The last kind of message that can be sent is an acknowledgement message. The first if statement (lines 161-189) is used to do two things. If this device is not the message's destination then we forward the message to the other device that the network is connected to,

assuming the routing table contains that route. We also append the current devices name to the *routingPath* variable so we can keep track of the route the message took as it went from destination to sender. This information is currently displayed in the terminal once the original sender receives the acknowledgement message. In future versions of the UI it would be nice to add the ability to see which devices the messages were routed through. Although, it should be noted this is only something you would want for testing as it is a potential security vulnerability allowing users to see the names of other user's devices. On the backend side, this information could be used for creating faster message delivery once mesh networks are implemented and the network is no longer linear.

The last area for this section we will look at are lines 191-199 where we determine the RTT and update the correct RouteUI object. Remember that these objects are used to determine if the device contains a given route. On line 201 we send a notification to the front end that tells the UI to update and display the new RTT we received.

Sending a Message:

The last specific implementation we will look at is how we actually send a message and interpret the different flags we pass it.

```
52      func send(text: String, with flag: Int, senderName: String, destinationName: String, routingPath: [String]) throws {
53
54          // Check to see if this device is sending a message
55          if flag == 0 && senderName == MPCManager.instance.localPeerID.displayName {
56              self.messageSendTime = Date.timeIntervalSinceReferenceDate
57              self.messageAckRecieved = false
58              logMessage(message: "Set message send time to \(self.messageSendTime)")
59          }
60
61
62          var routingInfoToSend: Set<String> = []
63
64          // Compile the routing table for this device and prepare to send
65          if flag == 1 || flag == 2 {
66              for device in MPCManager.instance.devices {
67                  if device != self {
68                      for route in device.routingInfo {
69                          routingInfoToSend.insert(route)
70                      }
71                  }
72              }
73              routingInfoToSend.insert(MPCManager.instance.localPeerID.displayName)
74          }
75
76          // If the device is sending an acknowledgement, we want to include this device
77          // in the route path back to the initial sender
78          var routePath = routingPath
79          if flag == 3 && !routePath.contains(MPCManager.instance.localPeerID.displayName){
80              routePath.append(MPCManager.instance.localPeerID.displayName)
81          }
82
83          let message = Message(body: text,
84                                flag: flag,
85                                routingInfo: routingInfoToSend,
86                                sendingDevice: senderName,
87                                destinationDevice: destinationName,
88                                routingPath: routePath)
89
90          let payload = try JSONEncoder().encode(message)
91
92          do {
93              try self.session?.send(payload, toPeers: [self.peerID], with: .reliable)
94          } catch {
95              throw error
96          }
97      }
```

For every message we use a *Message* object to store the data that we want to send. Depending on the type of message, some fields will be set to *null* or left empty.

Each Message objects contains the following:

1.) Message body
    - The words of the message the user wants to send
2.) Flag
    - Used to identify the type of message
3.) Routing Info
    - Contains routing info to be used by the receiving device to update their routing table
4.) Sending Device
    - The name of the device that sent the message
5.) Destination Device
    - The name of the device the message is being sent to
6.) Routing Path
    - The path routes the message passed through to get from the sending device to destination device.

Lines 54-59 are where we handle the first flag of 0 (Standard Message). If this device is the sending device and this is a standard message, we record the time the message is being sent so we can use the time to calculate the RTT once the ack message is received.

Lines 62-74 are used to load the *Routing Info* portion of the *Message* with the devices current routing table, plus a direct route to itself since we want the receiving device to be able to send messages back. Flags 1 (Routing Info Update) and 2 (Routing Info Request) both act the same when a message is being sent since they both send the current devices routing table.

The acknowledge message is handled on lines 76-81 by adding the current devices name to the *Routing Path*. This is used once the original sender receives the message so they can see the route the transmission traveled through.

Finally on lines 84-96 we bundle the different components into a *Message* object and send it to the device we are connected to.

6. Running the Project

Important: This project can only be run **on a Mac using macOS 10.14.4 or later**

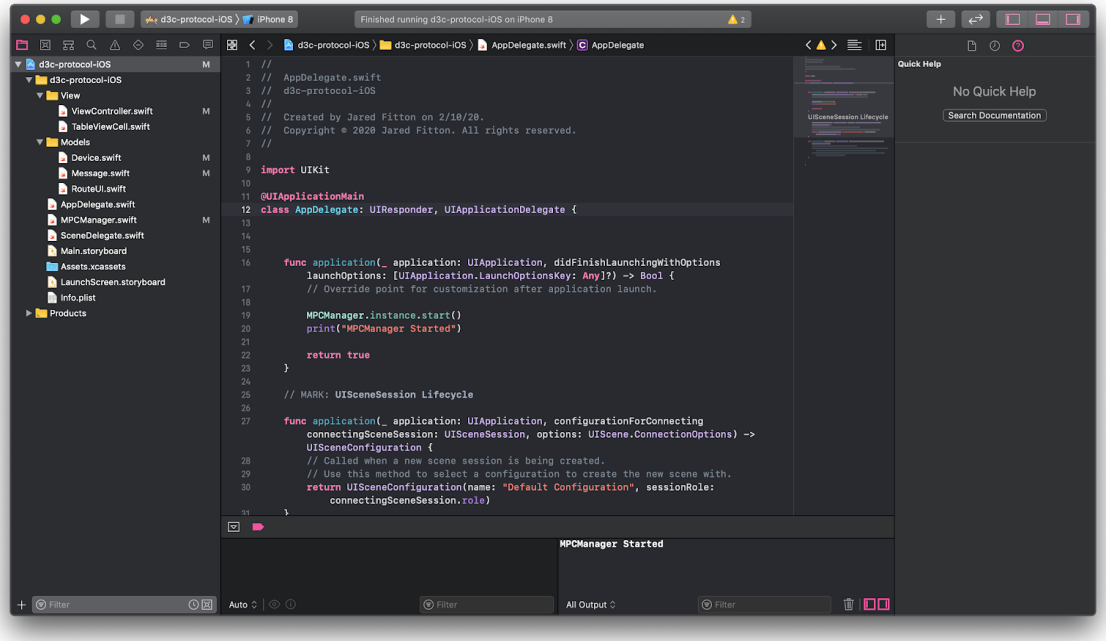
Before we can run the project, we need to get some of the dependencies setup first.

1.) Download and Install Xcode on a macOS device
    a.) This is needed to compile the code and run the project on an iOS device simulator

b.) Download link: https://developer.apple.com/xcode/resources/
c.) It will take awhile for it to download so don't worry if it is taking a long time
d.) Once it finishes installing you are ready for the next step

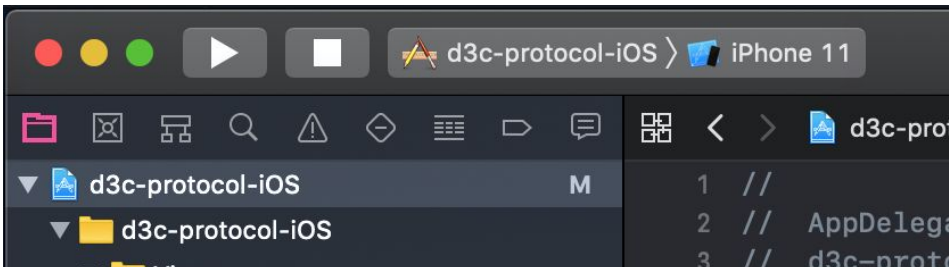
2.) Clone or download the code from the Github repository
a.) Download link: https://github.com/jaredfitton/d3c-protocol-poc

3.) Open the project
a.) Unzip the *d3c-protocol-poc* file
b.) Open the resulting folder named *d3c-protocol-poc*
c.) You should see a *README.md* file and another folder named *d3c-protocol-iOS,* open the folder
d.) Open the file named *d3c-protocol-iOS.xcodeproj*
e.) At this point the xcode project should be open and look like this:

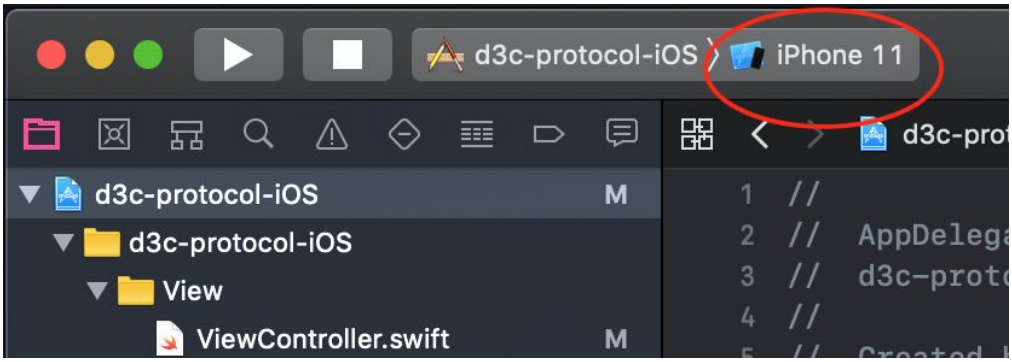


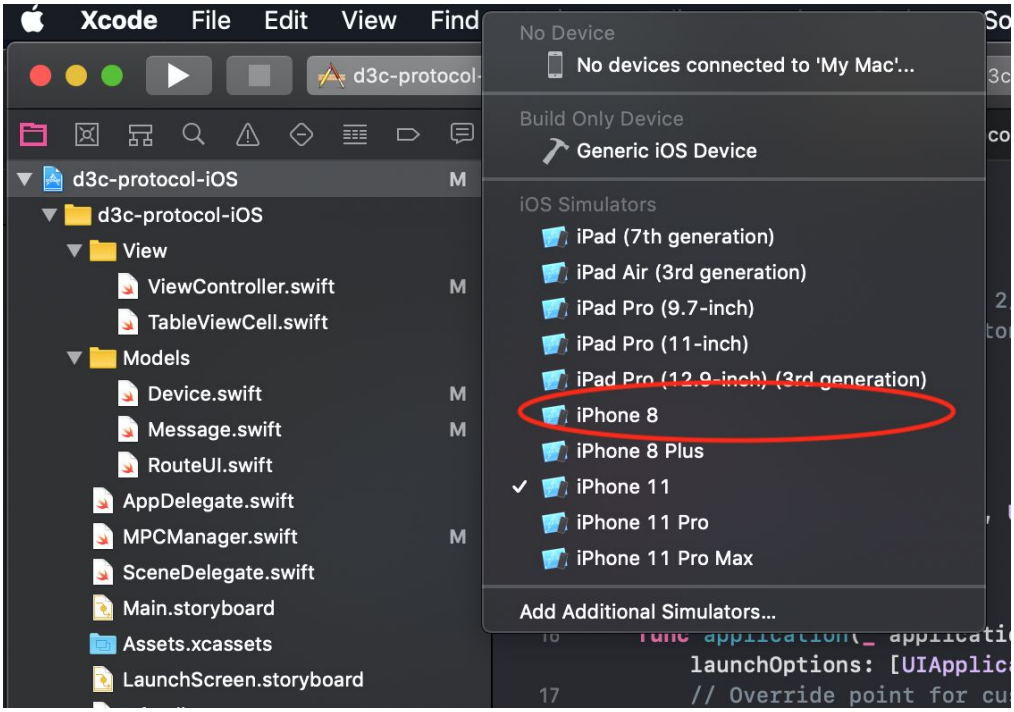
If you aren't on this exact file that is okay, all that matters is the project is open.

4.) Run the project
a.) When xcode runs a project, it does so by creating a simulator for the specified iOS device and then loading the app onto it.
b.) Due to the strain of running multiple simulators on your computer, I would recommend running only 3.
c.) These next steps are very important so follow them carefully. We are going to be launching 3 simulators in sequence.
i.) In the top left corner of xcode you should see a play button and an iOS device name.

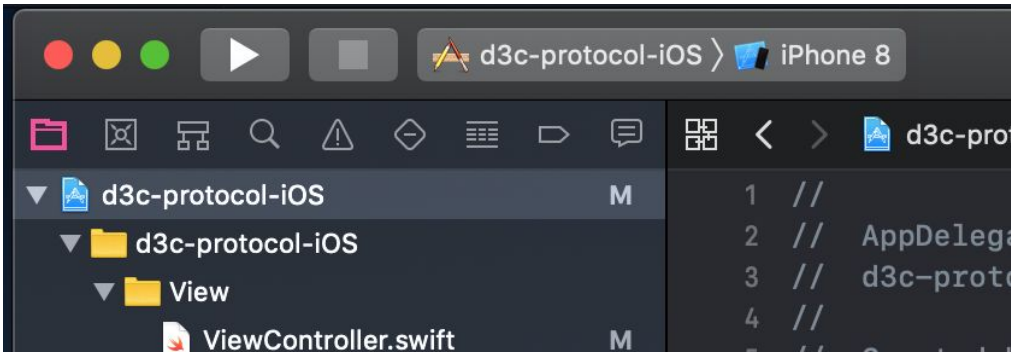


ii.) The device for you may not be the same as the above picture, but that is okay. We are going to go in and change it. Click on where it says “iPhone 11” and you will be presented with a drop down of iOS devices. Click the iPhone 8.

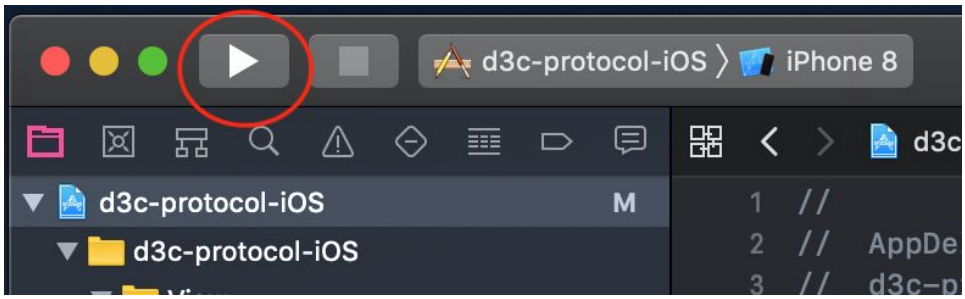




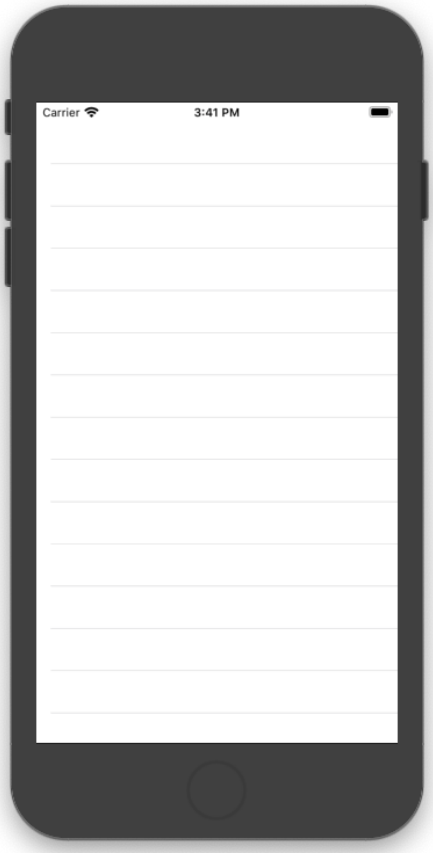
It should now look like the following:



iii.) Now that we have selected the first device, we will run the simulator. Do this by pressing the big play button.

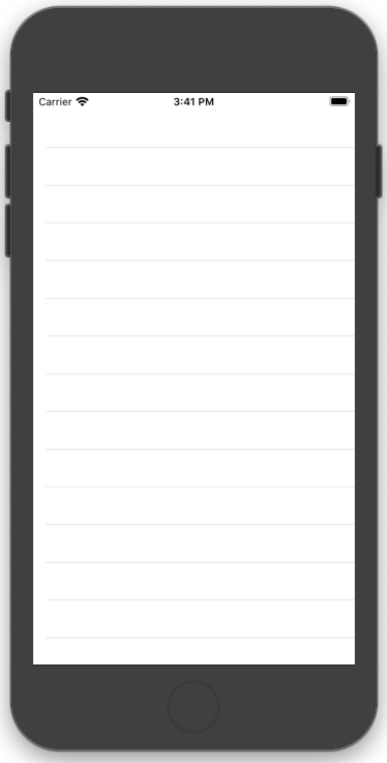


After doing so, the project will build, compile, and then launch a simulator. The simulator might take a minute or two to startup but once it has finished launching it should look like this:



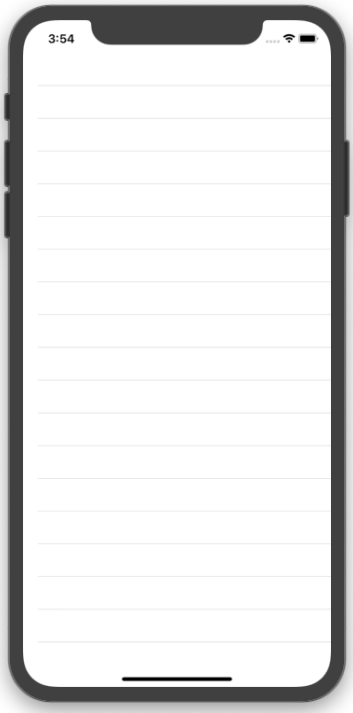
iv.) Now we will want to launch the next simulator. When we launch the next simulator, the app will close on the iPhone 8. Don't worry, this is supposed to happen. Repeat step 2-ii but instead of selecting the iPhone 8, select the iPhone 8 Plus.

v.) Run the simulator again like we did in step 2-iii. Allow the simulator time to launch. Once it is finished it should look like the following:



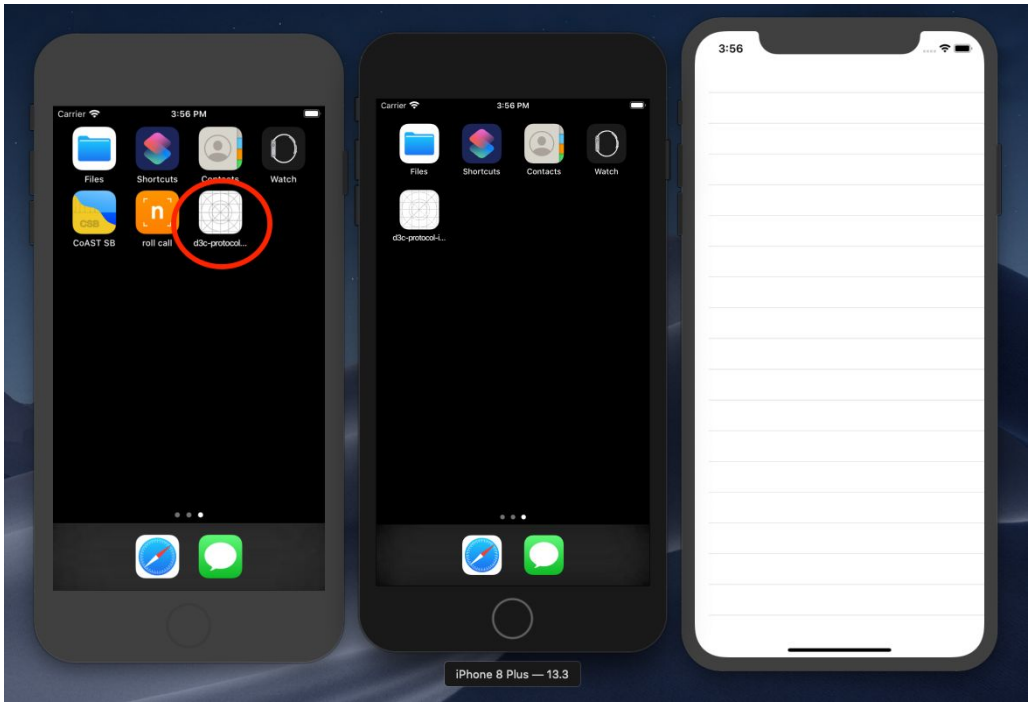
vi.) At this point we have a simulator for the iPhone 8 and the iPhone 8 Plus. We want to get one more simulator so we can see the device chaining in action. Remember that when we launch the simulator, the app will close on the iPhone 8 Plus. This is okay. Repeat step 2-ii but instead of selecting the iPhone 8 Plus, select the iPhone 11.

vii.) Run the simulator again like we did in step 2-iii. Allow the simulator time to launch. Once it is finished it should look like the following:

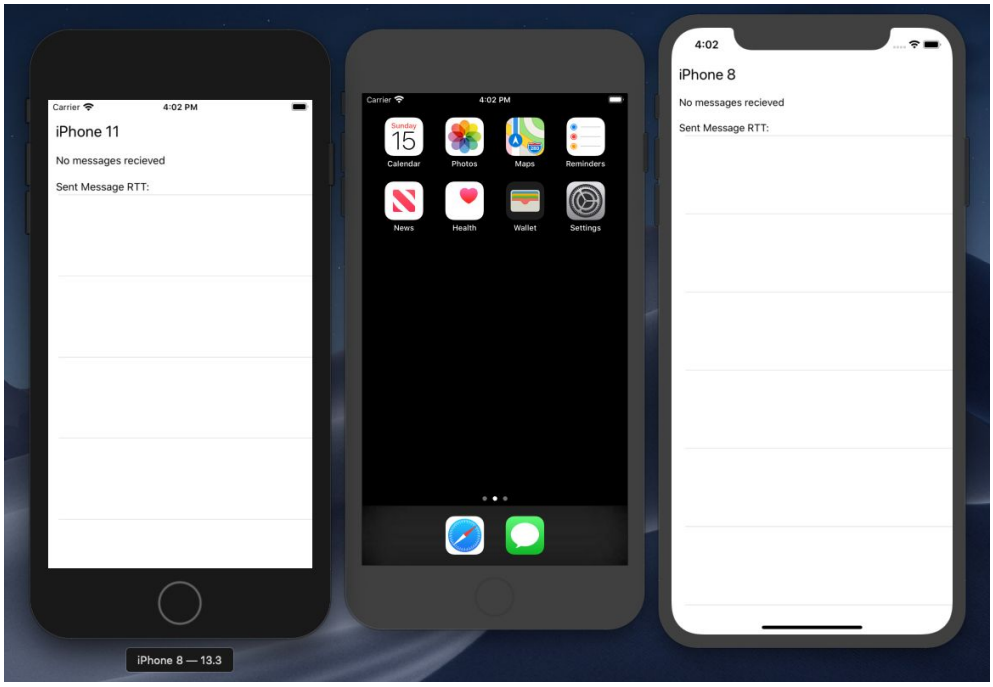


viii.) We now have three simulators running. If we look at all of them they should look something like the following:

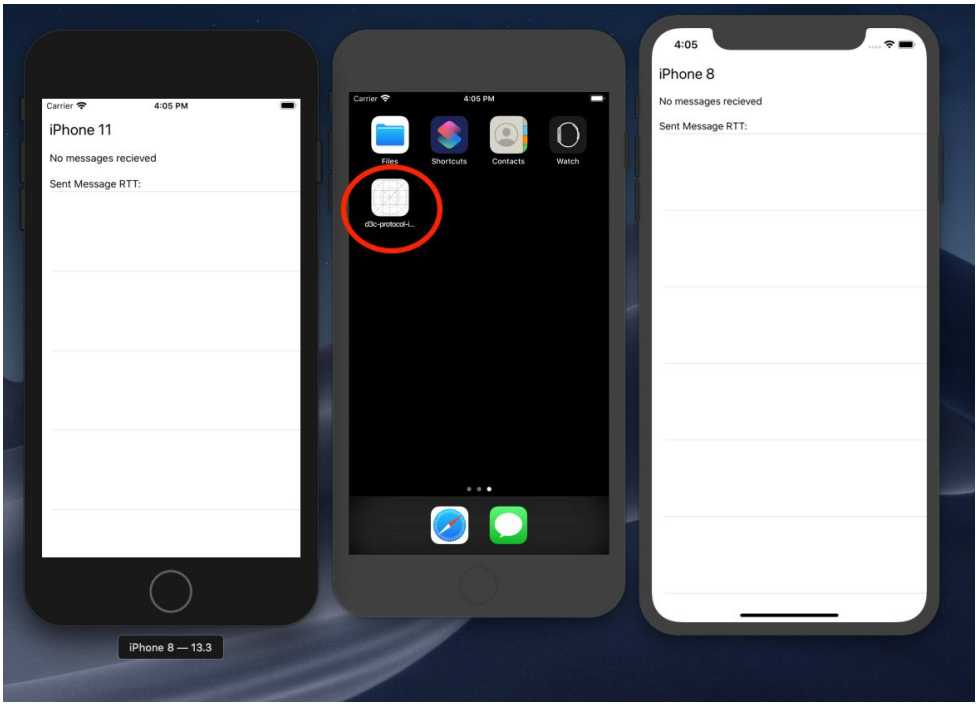
ix.) Now it is time to launch the app on the other devices. On the iPhone 8 (The phone farthest to the left in the above picture) click the app with the name “d3c-protocol…”. It will have a white icon with lines on it.



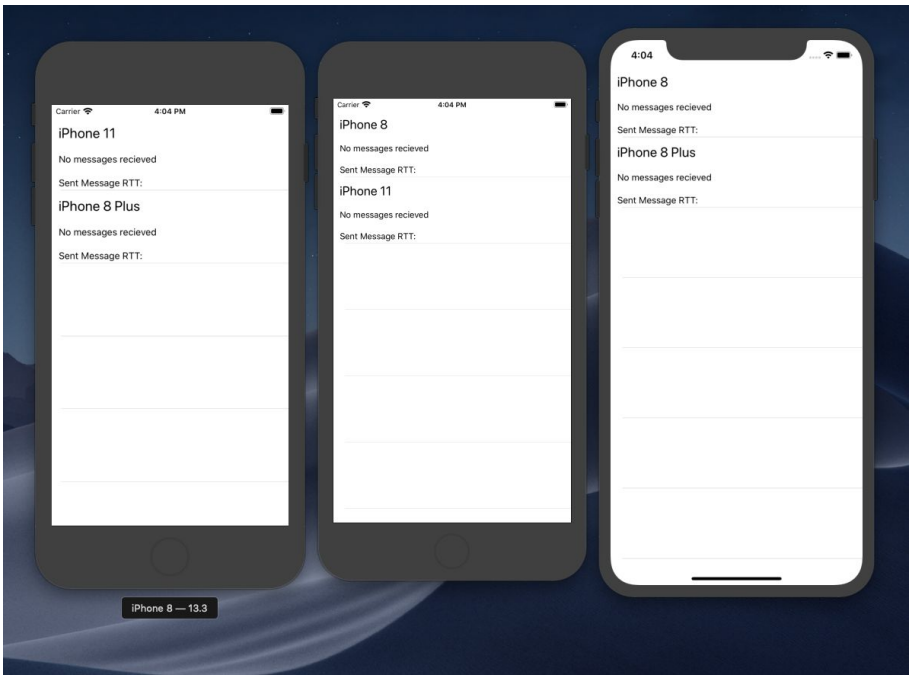
x.) After it launches you should see a connection made between the iPhone 11 and iPhone 8.



xi.) Finally, launch the app again on the iPhone 8 Plus.



xii.) After the app launches, all three devices should be able to access each other.

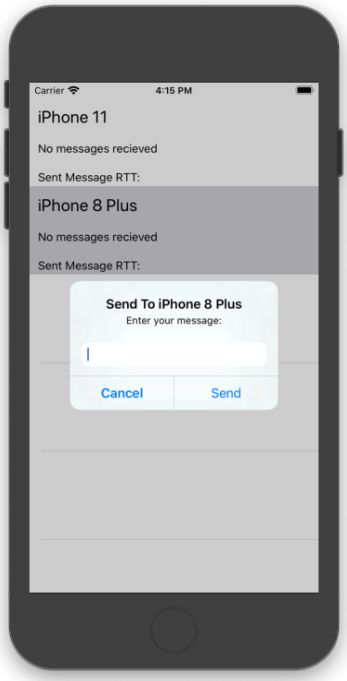


xiii.) You are now ready to begin using the app.

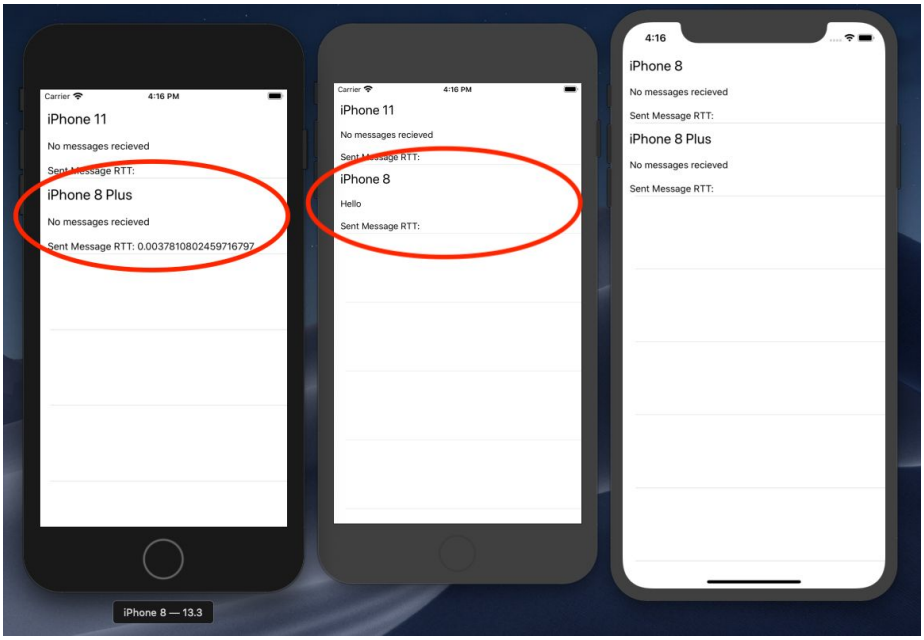
5.) Using the app

a.) Each iPhone lists the devices available through their routing tables.

b.) Tap on a device on one of the simulators to be prompted with a dialogue where you can enter a message that will be sent to the selected device:



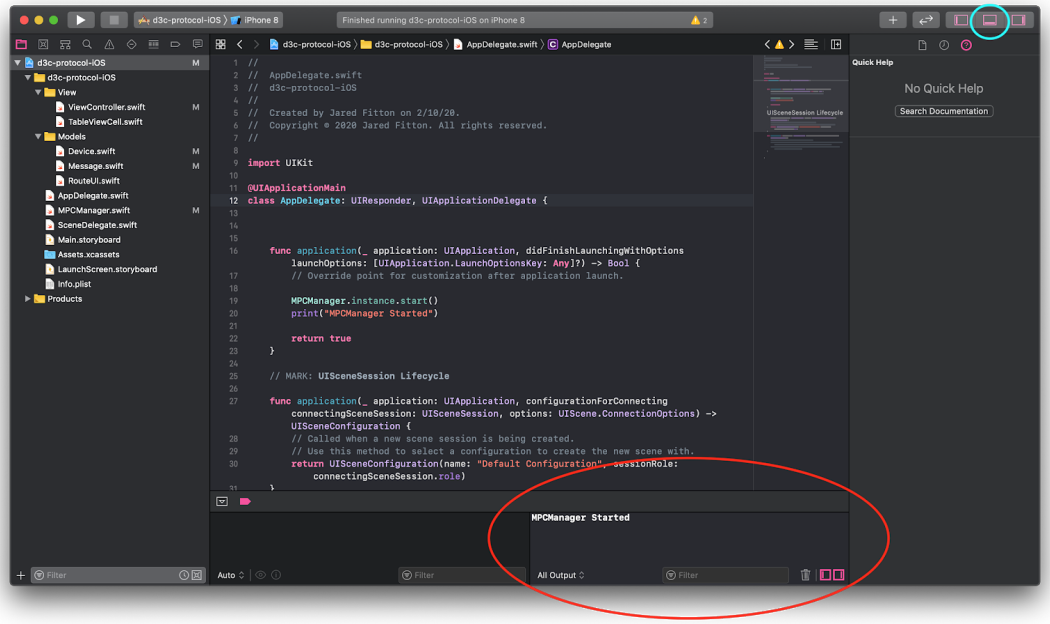
c.) Enter in your message and press send. Notice the message being received on the iPhone 8 Plus and the round trip time (RTT) being updated on the iPhone 8.



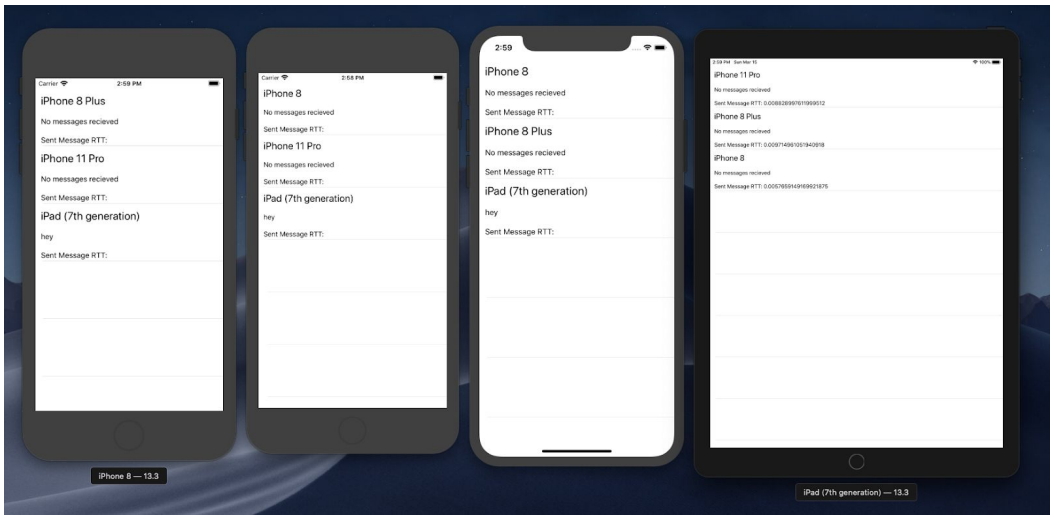
d.) From this point on feel free to play around with the app as you are all setup and ready to go. If you want to see the terminal output for the device that Xcode is currently checking, you can see at it the bottom of the Xcode window:
The red circle is the terminal. If you cannot see the terminal then make sure the bottom bar button is selected in the blue circle in the upper right hand corner.

7. UI Design

Since the main focus of this project was to make progress on the networking aspect, we decided to implement a basic UI that would be functional enough to demonstrate the messaging aspect of our project. On opening the app, it will display a screen that will have a different row for all possible messaging recipients which can be seen in section vii of the Implementation part of our report. This will be updated as new devices are added to your devices routing table which happens automatically when the devices come in range of each other. Each recipient is currently denoted using their device name but one of the features that could be actualized later on is allowing users to select their own usernames so that it would be easier to recognize the people that can be contacted. Clicking on a user creates a pop up box in which a user can input the message that they would like to send and an example can be seen in part 4b of the previous



section. The most recent message that the user received will be displayed below their name together with the RTT for the last message sent to the recipient, which can be seen in part 4c of the previous section. This can be expanded upon so that the whole message history will be displayed along with having the input box for the message to be sent on the same screen so that the app will have more of a messaging like feel that users are more familiar with.



7. Performance Evaluation

The majority of our time was spent on trying to implement the routing exchange and sending protocols and as such we didn't have enough time to do a performance evaluation. Since our application is basically designing a network and routing protocol, an important metric that we can use to analyze our network would be delay. There are two methods which we can use to look at the delay within our network. The first can be done by putting a timestamp on the message that has been sent and then when the recipient receives it, it will automatically check its own time and then calculate the difference between the two which we will then record. The second method that is useful in measuring delay is measuring round trip times. Here, we will once again record the time at which the message is sent, but instead of the recipient accounting for the difference the recipient will immediately send back an acknowledgment to which the original device. It will then determine what time it received it and subtract the difference from when it sent it to be able to calculate what the RTT is. There are numerous variables that can be altered while running tests and can be tested in combination with each other which include the size of the packets sent, the number of devices on the network, and the amount of additional traffic being sent to a device that is acting as a router at the same time that the original message is being sent. One of the last things that we were able to implement was RTT for messages and since it was one of the last features that we added we did not have enough time to gather data. Given more time, we would be able to perform more rigorous tests and simulations that would give a more accurate depiction of how our project would work in a more realistic network. Additionally, since our project is based on creating a network of devices, it requires numerous devices to be used so in our case we were simulating each device that was on the network. Collecting data from this would not have been representative of how it reacts in real life because the simulators do not behave the same as actual iOS devices, especially when it comes to their networks and frequency transmission. For testing, we would ideally like to have a collection of physical iOS devices that we can use as it will give us a better idea of how it would react to noise and physical latency. Another interesting comparison that we can do is tracking the time

that it takes to send a message using a cellular or WiFi network, and then comparing how much time it takes to reach a device versus using our peer to peer protocol. Bluetooth/peer-to-peer wifi isn't the best performing medium for data transfer but the main point of this project is as a proof of concept.

8. Future Features and Modifications

One feature that we would be interested in adding to our current protocol would be allowing devices to be able to connect to the network without being at the edge. Following that, we would also like to incorporate being able to remove devices from the network. We have already established how the protocol would implement it, however we just did not have enough time to code it.

The protocol for removing devices from the network first begins with the disconnecting device sending a message to the devices it is connected stating the current device is leaving. Following that, the disconnecting device can leave the network and each device that it was connected to will remove the disconnected device from their routing table. The original peering devices will then message the other devices they are connected to commanding them to remove the disconnected device from their routing lists. The devices that receive the command will do the same time and this command will propagate until it reaches one of the edge devices during which it will stop. At this point the device will have been removed from the network. Another feature we would like to add to our implementation would be to have a device join at the middle of two disconnected networks to bridge the gap so that it would "stitch" the two networks together. Since our app only works given that we uphold the invariant that we add devices linearly, we would like to greatly expand on our current networking protocol so that we would be able to create a full mesh network which is more representative of how our protocol would be used in real life. Additionally, we would also be interested in adding end-to-end encryption between devices since every device acting as a switch has access to the traffic and as such can snoop on the messages. Our app currently features a very rudimentary UI and as such an upgrade would definitely be in order for the users of the app to have the much needed functionality of being able to view a complete message history.

9. Conclusion

This project has helped us to better understand the problems that could occur when creating and implementing a new routing protocol. It was interesting to see that there could be an actual application for this project given that the mediums for a PAN increase to speeds fast enough to be competitive with cellular and WiFi speeds. This kind of technology could also be especially useful in natural disaster situations, where cellular/wifi isn't available, if it is implemented on the operating system side of the device. There are hundreds of millions of smart devices out in the world and, while adding this protocol to all of them isn't realistic, it is possible they can be updated since it uses existing technology and is built upon the software.

We would highly recommend others and our own group to continue working on this project as it has great potential to impact the world once it is fully implemented.