TRiGlobe Part 5 – Exchanging Data Between Devices

# P2P

We will use iOS7’s new Multipeer Connectivity Framework to communicate between two devices. We will only be exchanging string data for now but you can exchange many other data formats.

Ok so we have our user’s list in a tableview. Users must be able to post data to the tag table. This is where the Tag Class comes in. Create a new NSObject subclassed file and name it Tag. Now replace its code with the following:

#import <Foundation/Foundation.h>

@interface Tag : NSObject {

NSNumber \*rglatitude;

NSNumber \*rglongitude;

NSString \*originUdid;

NSString \*destintyUdid;

NSString \*rgcountry;

NSString \*sender;

NSString \*receiver;

}

@property(nonatomic,copy)NSString \*destintyUdid;

@property(nonatomic,copy)NSNumber \*rglatitude;

@property(nonatomic,copy)NSNumber \*rglongitude;

@property(nonatomic,copy)NSString \*originUdid;

@property(nonatomic,copy)NSString \*rgcountry;

@property(nonatomic,copy)NSString \*sender;

@property(nonatomic,copy)NSString \*receiver;

-(id)initWithOriginUdid:(NSString\*)oudid

rglatitude:(NSNumber\*)lati

rglongitude:(NSNumber\*)longi;

-(id)initWithSender:(NSString\*)senderi

receiver:(NSString\*)receiveri

rglatitude:(NSNumber\*)lati

rglongitude:(NSNumber\*)longi

rgcountry:(NSString\*)rgcountri;

-(id)initWithOriginUdid:(NSString\*)oudid

destintyUdid:(NSString\*)dudid

rglatitude:(NSNumber\*)lati

rglongitude:(NSNumber\*)longi

rgcountry:(NSString\*)rgcountri;

@end

#import "Tag.h"

@implementation Tag

-(id)initWithOriginUdid:(NSString\*)oudid

rglatitude:(NSNumber\*)lati

rglongitude:(NSNumber\*)longi

{

NSLog(@"TAG INIT");

if ( (self = [super init]) == nil )

return nil;

self.rglatitude = lati;

self.rglongitude = longi;

self.originUdid = oudid;

return self;

}

-(id)initWithSender:(NSString\*)senderi

receiver:(NSString\*)receiveri

rglatitude:(NSNumber\*)lati

rglongitude:(NSNumber\*)longi

rgcountry:(NSString\*)rgcountri

{

NSLog(@"TAG INIT OF TYPE SENDER RECEIVER");

if ( (self = [super init]) == nil )

return nil;

self.receiver = receiveri;

self.rglatitude = lati;

self.rglongitude = longi;

self.sender = senderi;

self.rgcountry = rgcountri;

return self;

}

-(id)initWithOriginUdid:(NSString\*)oudid

destintyUdid:(NSString\*)dudid

rglatitude:(NSNumber\*)lati

rglongitude:(NSNumber\*)longi

rgcountry:(NSString\*)rgcountri

{

NSLog(@"TAG INIT");

if ( (self = [super init]) == nil )

return nil;

self.destintyUdid = dudid;

self.rglatitude = lati;

self.rglongitude = longi;

self.originUdid = oudid;

self.rgcountry = rgcountri;

return self;

}

@end

Ok so we have a class that creates Tag objects and we have given it different initializers. So every time a user wants to create a tag they will have to use one of these methods. Users will create tags by bumping phones. This means each time users bump their phones they will be able to exchange tags. That is where Multipeer Connectivity framework comes in.

The grand scheme will be:

1. Add the Multi-peer Connectivity Framework to our project
2. Add code to create a connection
3. Add the code to exchange data

The first part is the easiest. Simply select the Blue Project folder atop your File Navigator on the leftmost pane. Now from the tabs that appear on the Main Editor Window, select Build Phases. Now click on the “+” button and type in Multipeer and select the Multipeer Framework. This will add the necessary files to your project.

Now onto step 2 where we add the code to make a multi-peer connection.

MULTIPEER CONNECTIVITY iOS7

In your AppDelegate.h add the following lines:

#import <MultipeerConnectivity/MultipeerConnectivity.h>

extern NSString \*const kServiceType;

@property (strong, nonatomic) MCSession \*session;

@property (strong, nonatomic) MCPeerID \*peerId;

extern NSString \*const DataReceivedNotification;

- (void)sendDataToPeer;

We are importing the proper framework, declaring a couple of constant strings, a few properties and the method that will be called to send the data to the other user. The two properties are required to make the connection; a session property and a peerID property.

In your AppDelegate.m add the following lines:

NSString \*const kServiceType = @"datashare";

@property (strong, nonatomic) MCAdvertiserAssistant \*advertiserAssistant;

NSString \*const DataReceivedNotification = @"com.santiapps.iGlobe:DataReceivedNotification";

We will use a string identifier for the service type, an advertiser assistant to advertise that we are running Multi-peer connectivity and an NSNotification identifier to call when data has been exchanged successfully. Finally add this protocol to the interface declaration.

<MCSessionDelegate>

Code to Exchange Data

Now that we have a way to connect, let’s work on the code for actually doing so and exchanging the data we need to exchange. First off, lets understand that our AppDelegate will manage the session and our MapViewController will initiate and end the data exchange. In order for the MapViewController to do those things, it needs to talk to the AppDelegate who will be managing the connection.

Ok so in the AppDelegate’s appDidFinishLaunchingWithOptions method, let’s add

// 1

NSString \*peerName = [[UIDevice currentDevice] name];

self.peerId = [[MCPeerID alloc] initWithDisplayName:peerName];

// 2

self.session =[[MCSession alloc] initWithPeer:self.peerId securityIdentity:nil encryptionPreference:MCEncryptionNone];

self.session.delegate = self;

// 3

self.advertiserAssistant = [[MCAdvertiserAssistant alloc] initWithServiceType:kServiceType discoveryInfo:nil session:self.session];

// 4

[self.advertiserAssistant start];

Here we are assigning a unique name to peerName, initting one of the peers to be used with that peerName and initting a session with that peer. We set the AppDelegate to be the delegate to the peer connection object.

Finally we create our advertiser assistant, which will advertise our availability when needed to other devices and we start it so we can be discoverable.

So the AppDelegate will be notified when important protocol events occur. Let’s add the methods required by this delegate protocol.

Just below the appDidFinishLaunchingWithOptions method add the following methods:

# pragma mark - Add bump-p2p

-(void)session:(MCSession \*)session didReceiveData:(NSData \*)data fromPeer:(MCPeerID \*)peerID{

NSString \*gotUser = [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];

NSLog(@"gotUser: %@", gotUser);

[[NSNotificationCenter defaultCenter] postNotificationName:DataReceivedNotification object:nil];

}

This method actually receives the data exchanged. Once we get it, we decode it in this case because we are simply getting an NSString which was UTF8 encoded. Once it is received we post a notification to the center reporting that we received it.

-(void)session:(MCSession \*)session didReceiveStream:(NSInputStream \*)stream withName:(NSString \*)streamName fromPeer:(MCPeerID \*)peerID{

}

-(void)session:(MCSession \*)session didFinishReceivingResourceWithName:(NSString \*)resourceName fromPeer:(MCPeerID \*)peerID atURL:(NSURL \*)localURL withError:(NSError \*)error{

}

-(void)session:(MCSession \*)session didStartReceivingResourceWithName:(NSString \*)resourceName fromPeer:(MCPeerID \*)peerID

withProgress:(NSProgress \*)progress{

}

-(void)session:(MCSession \*)session peer:(MCPeerID \*)peerID didChangeState:(MCSessionState)state {

}

These delegate methods are used for more advanced features that we don’t use in this tutorial. They are included here so you can see the many features available to us as developers when receiving data.

-(void)sendDataToPeer {

NSUserDefaults \*defaults = [NSUserDefaults standardUserDefaults];

if ([defaults objectForKey:@"storedUser"]) {

NSData \*myUser = [[defaults objectForKey:@"storedUser"] dataUsingEncoding:NSUTF8StringEncoding];

NSError \*error;

[self.session sendData:myUser toPeers:[self.session connectedPeers] withMode:MCSessionSendDataReliable error:&error];

} else {

NSLog(@"ERROR");

}

}

In the end, sendDataToPeer method takes the storedUser object from NSUserDefaults, encodes it into NSData and sends it to the other peer.

Ok so who calls these methods? The class we want to invoke these methods is the one the user will be interacting with when he decides to share the data with a user.

So in our MapViewController class lets add the following:

#import <MultipeerConnectivity/MultipeerConnectivity.h>

<MCBrowserViewControllerDelegate>

Once again we import the Multi-peer connectivity framework and adopt the protocol.

Now we add the methods that will be called, perhaps at the touch of a button:

#pragma bump-p2p

-(void)viewWillAppear:(BOOL)animated{

[super viewWillAppear:animated];

[self dataReceived:nil];

}

-(void)browserViewControllerWasCancelled:(MCBrowserViewController \*)browserViewController{

[browserViewController dismissViewControllerAnimated:YES completion:nil];

}

-(void)browserViewControllerDidFinish:(MCBrowserViewController\*)browserViewController{

[browserViewController dismissViewControllerAnimated:YES completion:^{

[self sendData];

[self showMessage:@"didFinish"];

}];

}

First we set up some delegate methods of what will happen when the browserViewController is cancelled or dismissed. The BrowserViewController will popup when the user taps for a connection. It will display the peers available for connectivity. It also requires a delegate, which will be the MapViewController itself. This should be familiar from working with Cancel and Done buttons.

-(void)sendData {

AppDelegate \*delegate = (AppDelegate \*)[[UIApplication sharedApplication] delegate];

[delegate sendDataToPeer];

}

To send data we need to call our delegate’s sendDataToPeer method.

-(void)showMessage:(NSString\*)message{

[[[UIAlertView alloc] initWithTitle:@"DEBUG" message:message delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil] show];

}

We want to keep the user notified of what is going on so we use a simple UIAlertView to present the user with information.

-(IBAction)sendButtonPressed:(id)sender {

AppDelegate \*delegate = (AppDelegate \*) [[UIApplication sharedApplication] delegate];

if ([[delegate.session connectedPeers] count] == 0) {

MCBrowserViewController \*browserViewController = [[MCBrowserViewController alloc] initWithServiceType:kServiceType session:delegate.session];

browserViewController.delegate = self;

[self showMessage:@"No Peers"];

[self presentViewController:browserViewController animated:YES completion:nil];

} else {

[self sendData];

[self showMessage:@"Peers: Sending Data"];

}

}

This is the main method in MapViewController regarding peer connectivity. It uses the AppDelegate to check if there are any peers available. If there are, then it creates a MCBrowserViewController in order to display them to the user. If there are no peers then it presents the correct UIAlertView, otherwise it shows a different message and begins sending data.

-(void)dataReceived:(NSNotification \*)notification{

[self showMessage:@"dataReceived called"];

}

Finally the dataReceived method is called when the data is received and it shows the appropriate method.

We must take care of some NSNotification set up and clean up. In viewDidLoad:

[[NSNotificationCenter defaultCenter] addObserver:self

selector:@selector(dataReceived:) name:DataReceivedNotification object:nil];

And in viewDidUnload we unregister:

[[NSNotificationCenter defaultCenter] removeObserver:self];

Perfect! Run this on your device and use a friend’s device to exchange data between phones or tablets.

We hope you enjoyed this long multi-part tutorial and drop us a line if you need any help!