APRICOT and SAFE

RnD report and Documentation

Bhaveshkumar Yadav
bhaveshy@iitb.ac.in
Guide: Prof. Bhaskaran Raman
Department of Computer Science & Engineering
IIT Bombay

June 22, 2020

1 Introduction

This report is the documentation of the work done on the SAFE and APRICOT app for iOS during the spring semester of 2019-20. SAFE (Smart, Authenticated, Fast Exams) is an application system used in IIT Bombay and 30+ other colleges for conducting exams, tracking attendance, online correction of regular exams, etc. APRICOT (A PRIvacy preserving COntact Tracing app) is a contact tracing app developed during the COVID-19 Pandemic to track the possible spread of infection. This documentation will help people pick up the project in the future. Section 2 describes the work done on Web Sockets for SAFE- iOS. Section 3 is about the periodic submission of a quiz in the background. Section 4 is the documentation of the iOS front-end APRICOT app, including details about Bluetooth scanning.

2 SAFE - Websocket in iOS

In the Android version of the SAFE app, during an ongoing quiz, a WebSocket connection is established with the back-end. This connection is used to send real-time information to the server. For example, when a student puts the app in the background, the corresponding flags are immediately sent to the server. Thus a teacher can view the real-time flags on the SAFE analytics dashboard. However, this feature is not implemented in the iOS app. The app logs information during the quiz locally. This information is only sent to the server when the quiz is submitted. Thus, while the quiz is going on, real-time data is not available from the iOS version of the app. My first task was to implement this feature in iOS. I started exploring various WebSocket libraries for iOS.

2.1 About WebSockets in iOS.

WebSocket was not a first-class citizen in iOS before iOS 13. Hence to implement WebSockets reliably and conveniently, we needed to use third-party libraries. One of the most popular libraries is **Starscream**. It does all the heavy lifting behind the scenes and makes it easier to manage WebSockets. The library is open source and is available on Github

However, from iOS 13 onwards, WebSockets became a first-class citizen and can be easily implemented using URLSession. But to support older versions of iOS using Starscream was necessary.

2.2 Starscream

2.2.1 Installation

- Add **pod 'Starscream', '~> 4.0.0'** at the end of the podfile
- Run **pod install** in the terminal

2.2.2 Usage

Import the library at the top of the swift file with `import Starscream`
Once imported, we can open a connection to a WebSocket server as follows:

```
var request = URLRequest(url: URL(string: "http://localhost:8080")!)
request.timeoutInterval = 5
socket = WebSocket(request: request)
socket.delegate = self
socket.connect()
```

Note that `socket` should be a class property defined at the beginning of class as `var socket:WebSocket!`
We are setting the current class as the delegate of the WebSocket class. Hence we must mention it in the class definition as follows:

```
class  YourClass : WebSocketDelegate {
  ....
}
```

After this, we must define the delegate methods to handle the WebSocket events. For example, the `didReceive` function receives all the WebSocket events. Here `isConnected` is a class property.

```
func didReceive(event: WebSocketEvent, client: WebSocket) {
  switch event {
  case .connected(let headers):
    isConnected = true
    print("websocket is connected: \(headers)")
  case .disconnected(let reason, let code):
    isConnected = false
    print("websocket is disconnected: \(reason) with code: \(code)")
  case .text(let string):
    print("Received text: \(string)")
  case .binary(let data):
    print("Received data: \(data.count)")
  case .error(let error):
    isConnected = false
    handleError(error)
  }
}
```

2.3 Using WebSocket in SAFE

I set up the WebSocket library using the above boilerplate code. Also, I confirmed the working of the library using echo WebSocket address `wss://echo.websocket.org` The WebSocket address used by the SAFE server is `https://guava.safe-analytics.in/ws/quizid/?token=token`, `quizid` and `token` need to be substituted in the URL. Quiz ID is available when a user starts a quiz. In iOS, the quiz ID is stored in `GlobalFN` class with the property name `quizid` and can be accessed directly as `GlobalFN().quizid` Token is also available in the `GlobalFN` class and can be accessed in the same way. The final URL can be generated as follows :

```
url  = "https://guava.safe-analytics.in/ws/"+
       GlobalFN().quizid+"/?token="+GlobalFN().token
```

These are the necessary steps required to set up WebSockets in iOS. However, the required feature was not completed because the WebSocket library could not connect to the WebSocket server. I tried another library called 'SwiftWebSockets' with no luck. Shahrukh Husain later completed the feature.

3 SAFE - periodic submission of quiz in the background

In the android version of SAFE, a quiz gets periodically submitted in the background so that data is not lost. In iOS, the quiz is only saved locally but not sent to the server. The task was to add this feature to iOS too. The periodic submission of the quiz happens through an HTTP post-call every 120 seconds.
The API endpoint for periodic submission is :

```
BASE_URL + "api/quiz/submission_key/partial_submission/"
```

`submission_key` is a unique key used for submitting a quiz.`BASE_URL` is the server URL. The function which submits the quiz is written in the file `QuestionViewController.swift`
It is named as `periodicSubmit()`. When a quiz begins, the function is set to be called periodically after every `PERIODIC_SUBMIT_INTERVAL_SECONDS`. The property is stored in the `GlobalFn` class and currently has a value of 120.
A timer is created in the `viewDidLoad` function of the `QuestionViewController` to call the function periodically. The timer is saved in a class property named `periodicSubmitTimer` so that it can be stopped later. The following line of code creates the timer.

```
self.periodicSubmitTimer =
Timer.scheduledTimer(timeInterval:
                     GlobalFN().PERIODIC_SUBMIT_INTERVAL_SECONDS,
                     target: self,
                     selector: #selector(self.periodicSubmit),
                     userInfo: nil, repeats: true)
```

Thus `perodicSubmit` function will be called every 120 seconds until the timer is invalidated.

Inside the `periodicSubmit` function an empty array of dictionaries is initialized named `submission`. This array is then passed to the function `makeJsonObjectOfResponse` which then creates the submission JSON to be sent with the API call.

```
var submission = [[String:AnyObject?]]()
makeJsonObjectOfResponse(submission: &submission)
```

The parameters are defined as follows:

```
var parameters : [String : Any] = [
                "seconds_since_mark" : "0",
                "quizData" : submission,
                ]
```

The code following this is the normal routine for making a POST call. With error and response handling. When the quiz ends or if the user submits the quiz, the autoSubmit function is called. All the periodic timers are invalidated in this function. The periodicSubmit timer created earlier is also invalidated using:

```
self.peridoicSubmitTimer.invalidate()
```

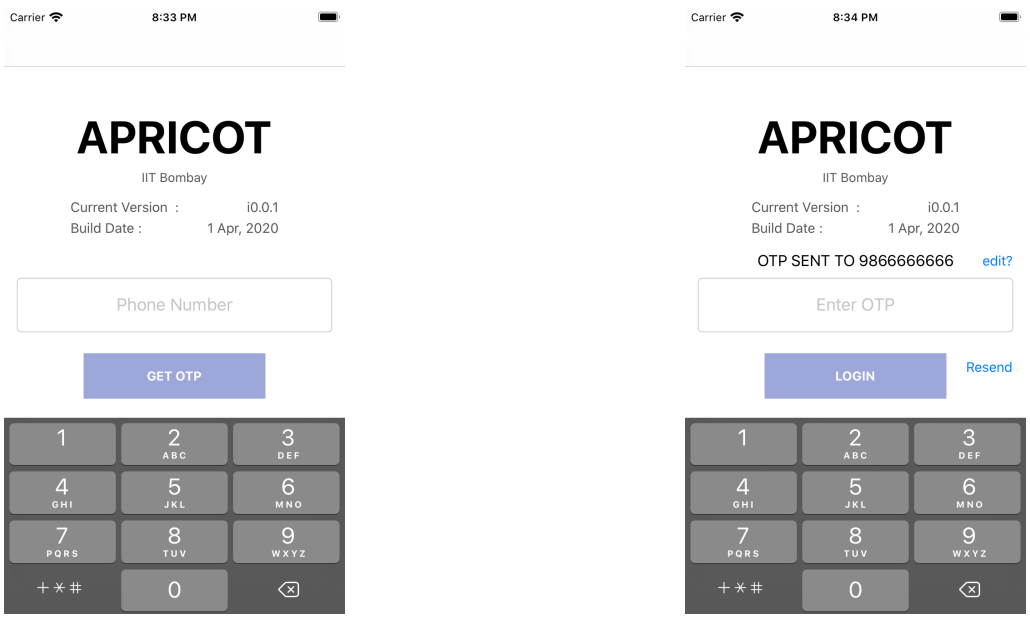
The feature is implemented in the branch `peridic_submit`

4 APRICOT (A PRIvacy preserving COntact Tracing app)

APRICOT is an open-source system developed under the guidance of Prof. Bhaskaran Raman at IIT Bombay during the corona pandemic. It is used to trace people who might have come in contact with a corona positive patient. The design of the system is based on guesses on how the contact Trace Together app works (Doc) The app uses Bluetooth and random ID's to keep track of devices that have come in close proximity to the user. The information, however, is kept confidential and stored locally on the user's device. This information can be uploaded to the server voluntarily by a user. APRICOT also has a web portal. The web portal can be used by authorities to view contact traces of people who have tested positive. This way they can warn the people who have come in contact with such patients. I worked on the front end of the app, specifically on iOS. The app is open-sourced and can be found on Gitlab.

4.1 Login Module

When the app is used for the first time by a user, the app asks the user to login using his phone number. The phone number is verified using a one-time-password. The login module is located in the `App Modules` folder. It has two important sub-modules. The `storyboard` and `interface`. The storyboard module has the `Login.storyboard` which is the UI file for the login screen. It is used to define the user interface. The `interface` module has the controller file named `LoginViewController.swift`. All the business logic for login is present in this file. The login screen view is very similar to the SAFE app login as shown in figure 1



(a) Phone Number Input

(b) OTP Input

Figure 1: The login Screen

The input field takes numbers up to 10 digits. Restricting the phone number field to 10 digits is done using the `phoneNumFieldDidChange` function. The function is set to be triggered whenever the input field changes using the following lines of code:

```
phoneNumberInput.addTarget(self,
                           action: #selector(phoneNumFieldDidChange),
                           for: UIControl.Event.editingChanged)
```

The `phoneNumFieldDidChange` function is as follows:

```
@objc func phoneNumFieldDidChange(){
    var phoneNumberCount = 0
    if let phoneNumber = phoneNumberInput.text {
        phoneNumberCount = phoneNumber.count
    }
    if(phoneNumberCount < 10) {
        getOTPButton.isEnabled = false
        getOTPButton.alpha = 0.5
    }else {
        if(phoneNumberCount > 10) {
            phoneNumberInput.deleteBackward()
        }
        getOTPButton.isEnabled = true
        getOTPButton.alpha = 1
    }
}
```

On touching the **GET OTP** button the function `getOTP()` is called. The `getOTP()` function stores the entered phone number in the `GlobalFn` class and then calls the `apiToGenerateOTP` function.

```
@objc func getOTP(){
    let phoneNumber = phoneNumberInput.text!
        GlobalFN().phoneNum =  phoneNumber
        apiToGenerateOTP()
    }
```

The `apiToGenerateOTP` function first shows a loading indicator using SwiftSpinner class.

```
SwiftSpinner.show("Sending OTP")
```

Then it prepares the parameters for the API call. The required parameters are:

- **`phone`:** Phone number entered by the user to which the OTP is to be sent
- **`device_id`:** This is a unique ID used to identify this particular device at the server.
- **`app_id`:** This is a unique ID used by the backend to identify a particular app version and on the OS which it is running.

Generating unique device ID in iOS

Early iOS releases provided a unique identifier for each device. But then this identifier was used by developers to uniquely identify a particular user. This was a security concern for Apple. Hence they changed the unique identifier with a vendor-specific device identifier. The identifier's value is the same for apps from the same vendor on the same device. But it's different for other vendors on the same device. The unique identifier is provided from the App Store. If the app is not present on the App Store then the identifier is derived from the bundle identifier of the App. Here is how to get the unique identifier.

```
let uuid = UIDevice.current.identifierForVendor?.uuidString
```

The UUID is going to be the same for every launch of the App. However, if the user uninstalls the app and re-installs it then the UUID will change. To keep the UUID the same even after re-install we should save the UUID in the keychain. The entries in keychain are kept even after the app is uninstalled unless the user clears them explicitly. The UUID is stored in a property named `keychainUniqueDeviceId` in the `GlobalFn` class. When the app starts for the first time this value is an empty string. In the `AppDelegate` class, this value is checked and if it's an empty string then the value is generated and saved in the keychain as follows:

```
if(GlobalFN().keychainUniqueDeviceId == ""){
    if let uuid = UIDevice.current.identifierForVendor?.uuidString{
        GlobalFN().keychainUniqueDeviceId = uuid;
    }
 }
```

Afterward, the `keychainUniqueDeviceId` is accessed from the `GlobalFn` class. The getter and setter are defined as follows:

```
var keychainUniqueDeviceId:String {
        get {
            let bundleName = Bundle.main.bundleIdentifier
            let keychain = Keychain(service: bundleName ?? "apricot")
            if let id = keychain[UniqueDeviceKey]  {
                return id
            }
            return ""
        }
        set {
            let bundleName = Bundle.main.bundleIdentifier
            let keychain = Keychain(service: bundleName ?? "apricot")

            if let t = keychain[UniqueDeviceKey]  {
                print("key already saved : "+t)
            }
            else {
                keychain[UniqueDeviceKey] = newValue

            }

        }
    }
```

The `app_id`:

The `app_id` is set as required by the back-end. It's a string used to identify a particular version of the app. After constructing the parameter dictionary as follows:

```
let parameters : [String : String] = [
        "phone" : GlobalFN().phoneNum as String,
        "device_id" :    GlobalFN().keychainUniqueDeviceId,
        "app_id" : GlobalFN().app_id
    ]
```

The dictionary is converted to JSON and then the appropriate format required by the API using,

```
let jsonString = GlobalFN().dictToJSON(dict: parameters as NSDictionary)
let jsonData = jsonString.data(using: .utf8)!
let attListRequest =
            try! JSONDecoder().decode(otpRequest.self, from: jsonData)
```

And finally, we make the API call using,

```
ApricotAPIManager.generateOTP(lastUpdateRQ:attListRequest)
```

The API end-point for generating OTP is:

```
    BASE_URL+"api/v1/login/generate-otp/"
```

This value of the BASE_URL as picked up from the ApricotNetworkURL.plist file. The callback function is written as a closure. The function is called when the API returns either success or error. If the API returns success then we load the OTP screen by calling the function:

```
self.waitForOTP()
```

The **self** keyword is important because we are calling the function from a closure.

The waitForOTP function does the following:

- Updates the OTP SENT TO text with the user's phone number and also makes it visible.
- Makes the editPhoneNumButton and Resend OTP button visible.
- Disables the Resend OTP button and sets a timer to enable it after OTP_MIN_INTERVAL seconds, stored in GlobalFn class. The timer is stored in the property otpTimer. It will be used to re-start or invalidate the timer.
- Changes the GET OTP button label to LOGIN and changes the function it triggers to checkOTPAndLogin
- Changes the placeholder of the input field to Enter OTP
- Changes the listener function for the phone number input to OTPInputDidChange
- Hides the SwiftSpinner popup.

After which the screen looks as shown in figure 1b
The input field used for the phone number is used for OTP too. The function OTPInputDidChange checks for the entered OTP and enables the LOGIN button when the OTP length reaches the OTP_LENGTH defined in GlobalFn class. Currently, this value is 6. This function is very similar to the phone number equivalent function but kept them different to improve readability.
After OTP_MIN_INTERVAL the user can click on the Resend? button and regenerate the OTP. The API used for generating the OTP is used for re-generating too. The user can click on the edit? button to edit the phone number and resend the OTP. The loadPhoneNumber function manages this. Once the user enters the six-digit OTP and clicks on LOGIN theverify-otp API is called from the function apiToVerifyOTP. The API call is similar to generate-otp and has only one extra parameter, the OTP entered by the user. The API end-point is:

```
"api/v1/login/verify-otp/"
```

If the OTP is correct the API returns a success message and the following:

- **token**: A unique login token to determine if the user is logged in.
- **user_id**: A unique ID maintained at the back-end.
- **pseudo_id_list**: A list of pseudo ID's to be used for Bluetooth scanning and broadcasting.

After receiving these, the values are stored in the User Defaults which is a persistent key-value storage for iOS apps to store small values. The getters and setters for these values are defined in the GlobalFn class.

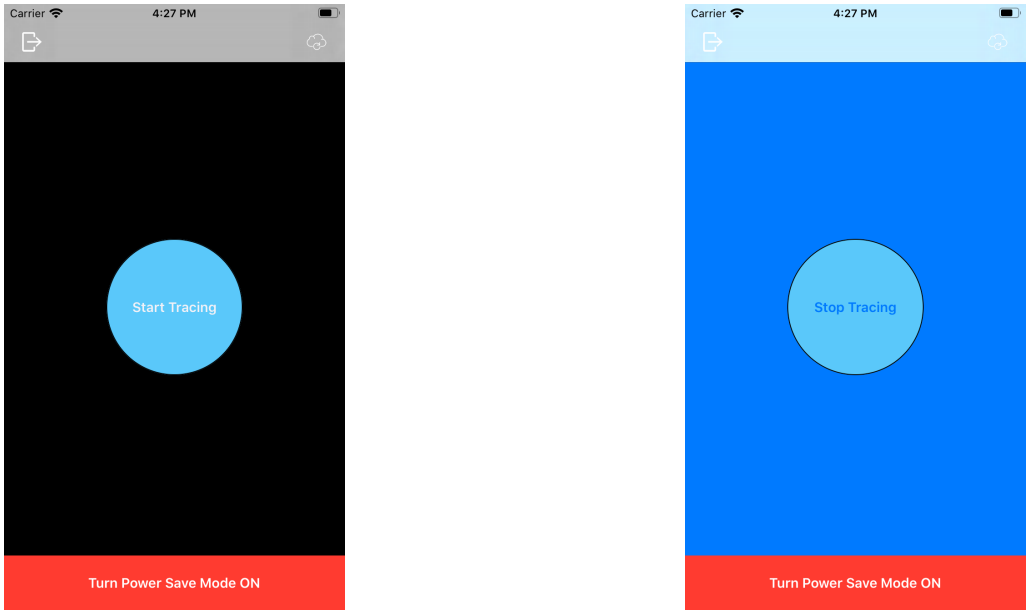
Saving the values:

```
GlobalFN().token = valueDict["token"] as! String
GlobalFN().user_id = valueDict["user_id"] as! String
GlobalFN().pseudo_id_list = valueDict["pseudo_id_list"] as! NSArray
```

Getters and Setters:

```
var token: String {
    get {
        if let returnValue = UserDefaults.standard.object(forKey: "token") as? String {
            return returnValue
        } else {
            return "" //Default value
        }
    }
    set {
        UserDefaults.standard.set(newValue, forKey: "token")
        UserDefaults.standard.synchronize()
    }
}
```

Storing the `user_id` is similar. The `pseudo_id_list` is an array of 1000 pseudo ID's. The device chooses one of them to broadcast, and it changes the ID every 5 minutes. This enables the privacy of the user. The array is stored in UserDefaults. So that once logged in, the array is directly read from the local storage without needing an API call. Since the pseudo ID list is an array, to retrieve it, we need to cast it as an `NSArray`.



(a) Tracing off

(b) Tracing on

Figure 2: The Main Screen

Once the user is logged-in, the main Bluetooth scanning screen is loaded with animation using:

```
let storyboard = UIStoryboard(name: "Trace", bundle: nil)
let vc =
 storyboard.instantiateViewController(withIdentifier: "TraceVC") as! TraceViewController
self.pushViewControllerWithTransition(vc: vc)
```

The `pushViewControllerWithTransition` function pushes the `TraceVC` view controller on top of the screen with a `flip` animation. The user then sees the screen in figure 2
On clicking`Start Tracing`. The app begins scanning for Bluetooth devices broadcasting their pseudo ID's nearby. It also broadcasts its own pseudo ID. The app then stores the pseudo ID of nearby devices with timestamp locally. Each phone scans for the nearby devices for a time of 1 min. The contact trace history is deleted after 30 days. Using the upload button, the user can upload his contact trace to the server. iOS uses iBeacon for Bluetooth beaconing. There is one challenge in iOS. We have a limited number of bits for pseudo ID (16 bits). This restriction is not present in Android. In the future, we plan to use Eddy Beacon and iBeacon both to increase this limit. Currently, if the user exits from the app, the app stops scanning. Hence we encourage users to keep the app in the foreground. That is why we provide a power-saving mode which reduces the brightness to zero.

5 Acknowledgement

I thank Prof. Bhaskaran Raman for giving me the opportunity to work on these interesting projects and also for his valuable guidance during the process.

References

[1] https://medium.com/better-programming/websockets-in-ios-using-swift-a176791e139f.

[2] https://github.com/tidwall/SwiftWebSocket.

[3] https://medium.com/better-programming/websockets-in-ios-13-using-swift-and-xcode-11-18fa3000d802.

[4] https://www.hackingwithswift.com/example-code/system/how-to-make-an-action-repeat-using-timer.

[5] https://developer.apple.com/documentation/uikit/uidevice/1620059-identifierforvendor.

[6] https://iitb-apricot.github.io/apricot-info/.