SWIFTCRAFT
21-24 May 2024

Bridging Apps with iOS Multipeer Connectivity

Zamzam Farzamipooya

Bridging Apps with iOS Multipeer Connectivity

Zamzam Farzamipooya
Engineering Manager @ Veo

SwiftCraft 2024

@zamzampooya

What Is Multipeer Connectivity

Support peer-to-peer connectivity and the discovery of nearby devices

In iOS, the framework uses

- infrastructure Wi-Fi networks,
- peer-to-peer Wi-Fi,
- and Bluetooth personal area networks for the underlying transport.

In macOS and tvOS, it uses

- infrastructure Wi-Fi,
- peer-to-peer Wi-Fi,
- and Ethernet.

Why Multipeer Connectivity?

Decentralized Network

No Internet Required

Ease of Use

Adaptive Connectivity

Versatility

Why Not?

Limited Range

Performance Variability

Complex Network Management

Compatibility

What It Can Be Used For

Collaborative tools

Messaging

Education

Location based

Social Networking

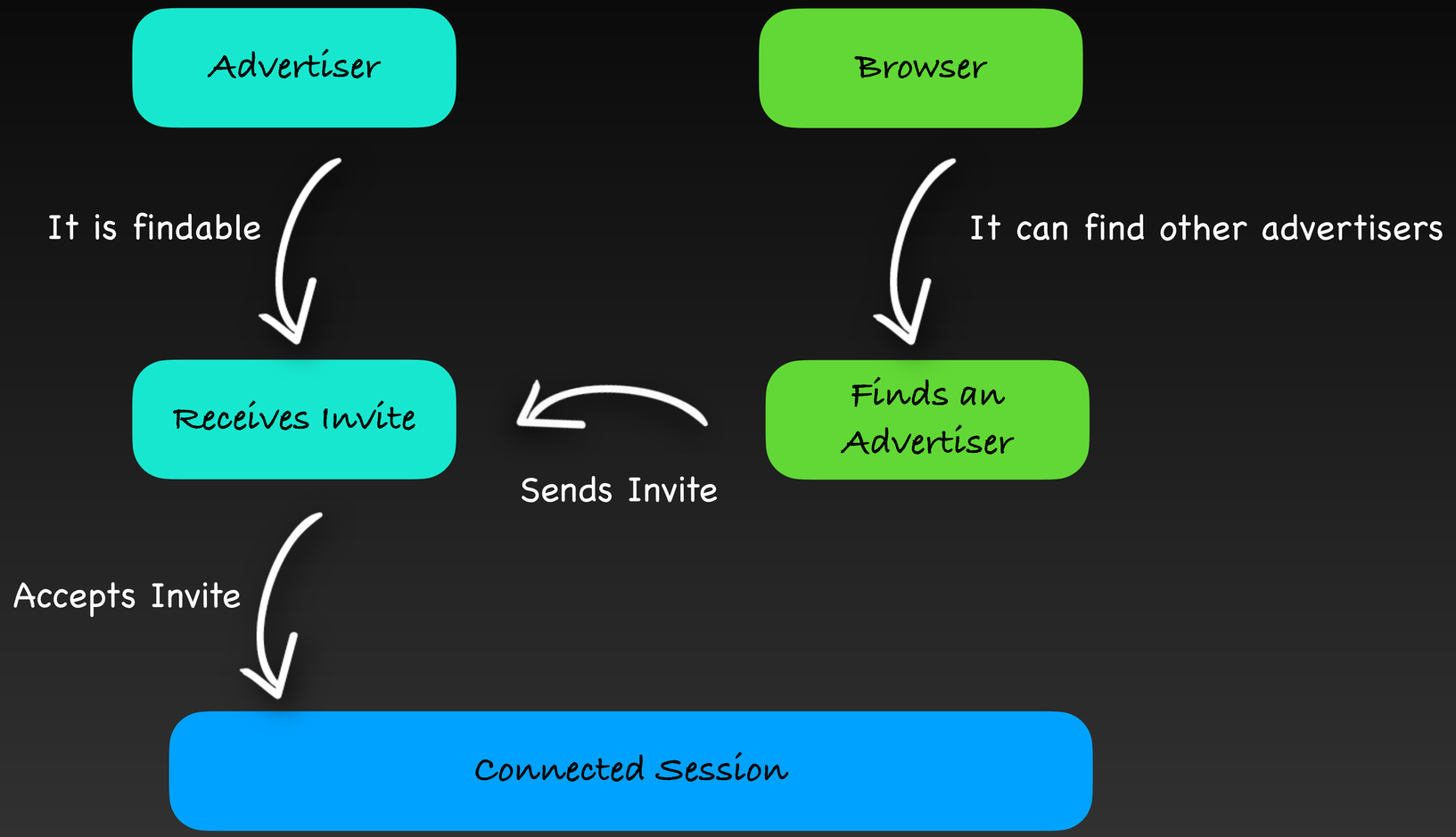
Gaming!

Multipeer Connectivity Framework

Key Concepts

- **Session objects (MCSession)** support communication between connected peer devices.
- **Advertiser objects (MCNearbyServiceAdvertiser)** tell nearby peers that your app is willing to join sessions of a specified type.
- **Browser objects (MCNearbyServiceBrowser)** let your app search programmatically for nearby devices with apps that support sessions of a particular type.
- **Peer IDs (MCPeerID)** uniquely identify an app running on a device to nearby peers.

How It Works



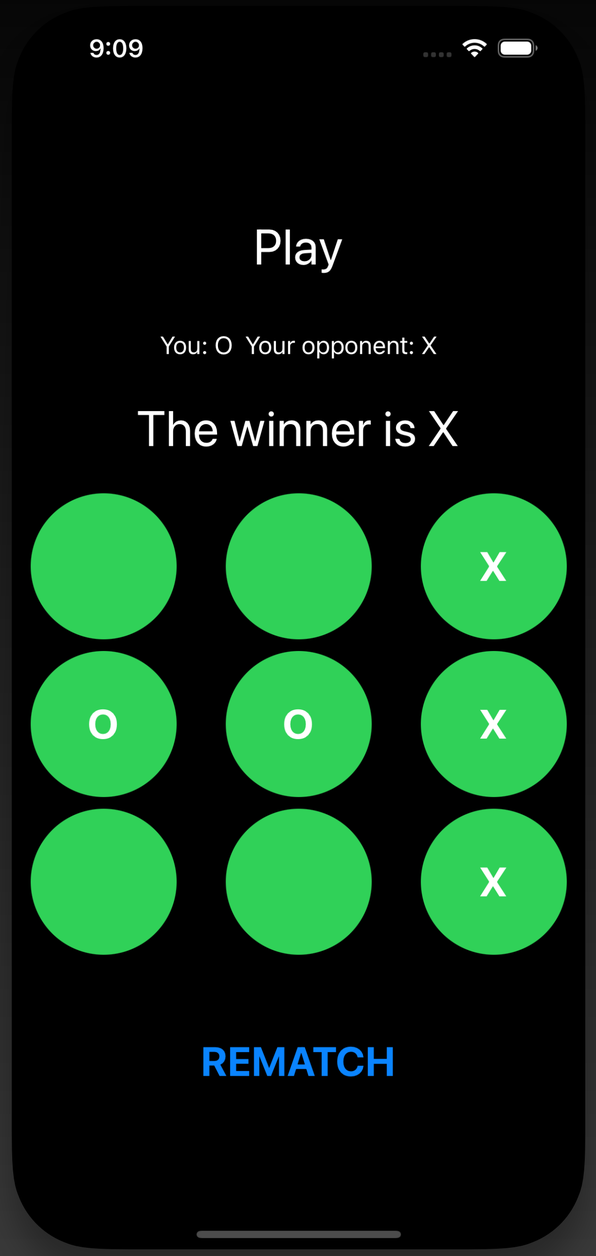
They can send and receive data

Tic-Tac-Toe

Find nearby players

Connect to one opponent

Play!

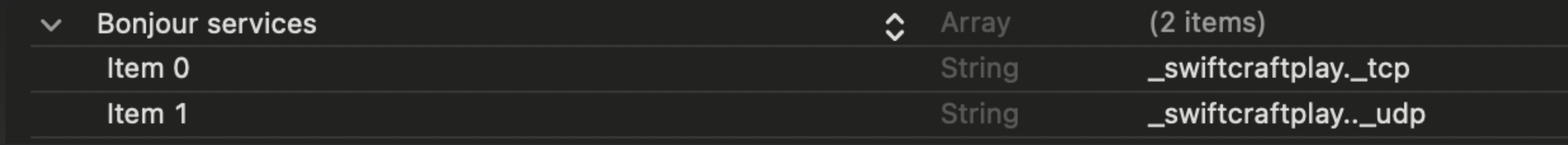


Setting Up Info.plist

Getting permissions from the user

Privacy — Local Network Usage Description

Also we need to add the following to Info.plist



MCManager

ServiceType -> The same name as in the info.plist

Advertiser

Browser

Session

Players

```
@Observable
class MCManager: NSObject {

    static let share = MCManager()
    let serviceType = "swiftcraftplay"
    var advertiser: MCNearbyServiceAdvertiser
    var browser: MCNearbyServiceBrowser
    var session: MCSession?
    var currentPlayer: Player
    var opponent: Player?
    var availablePlayers: Set<Player> = []
```

Player

A model to keep track of players

MCPeerID

Name

MCSessionState

PlayerType

```
struct Player: Hashable, Identifiable {
    var id: MCPeerID
    var name: String
    var state = MCSessionState.notConnected
    var playerType: PlayerType?
```

```
enum PlayerType: String, Codable {
    case x = "X"
    case o = "O"
}
```

Setting Up The Environment

- PeerID
- CurrentPlayer
- Session
- Advertiser
- Browser
- Their delegates

```
override init() {
    let peerID = MCPeerID(displayName: solarSystems.randomElement() ?? "Sol")
    currentPlayer = Player(id: peerID, name: peerID.displayName)
    session = MCSession(peer: peerID, securityIdentity: nil, encryptionPreference: .none)
    advertiser = MCNearbyServiceAdvertiser(peer: peerID, discoveryInfo: nil, serviceType:
        serviceType)
    browser = MCNearbyServiceBrowser(peer: peerID, serviceType: serviceType)

    super.init()


    session?.delegate = self
    advertiser.delegate = self
    browser.delegate = self
}
```

Starting The Sessions

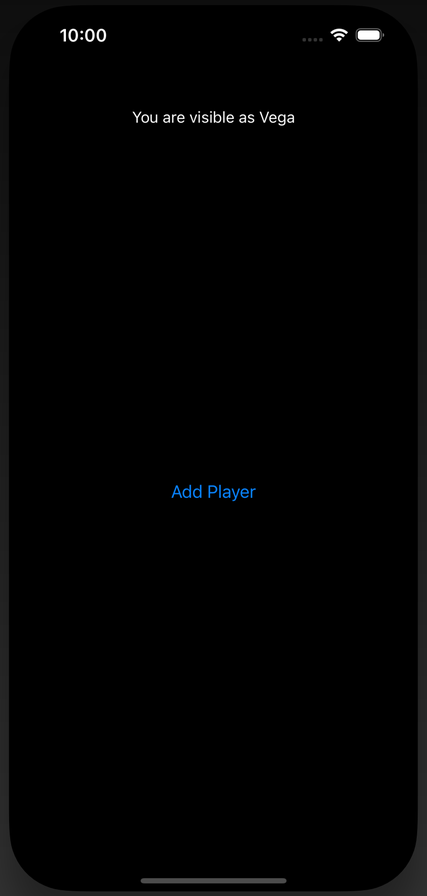
```
struct ContentView: View {

    @State var mcManager = MCManager.share

    var body: some View {
        VStack {
            switch mcManager.viewState {•••}

        }.onAppear {
            mcManager.start()
        }
    }

}
```

```
enum ViewState {
    case addPlayer
    case findingPlayers
    case play
}
```

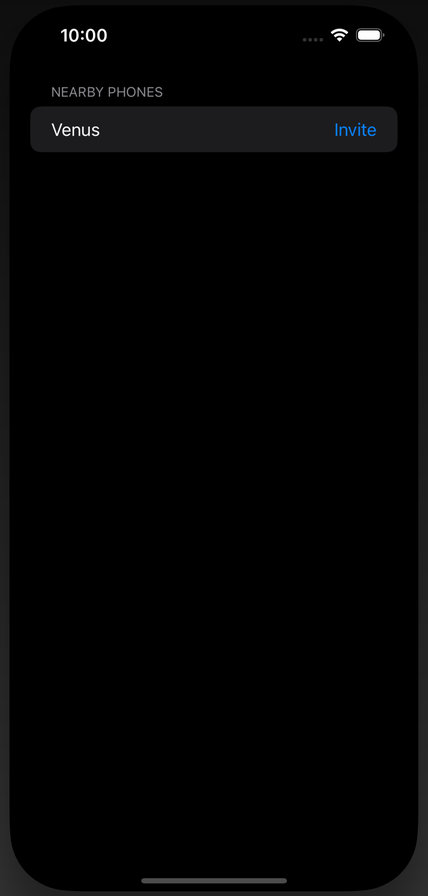
```
func start() {
    advertiser.startAdvertisingPeer()
    browser.startBrowsingForPeers()
}
```

Starting The Sessions

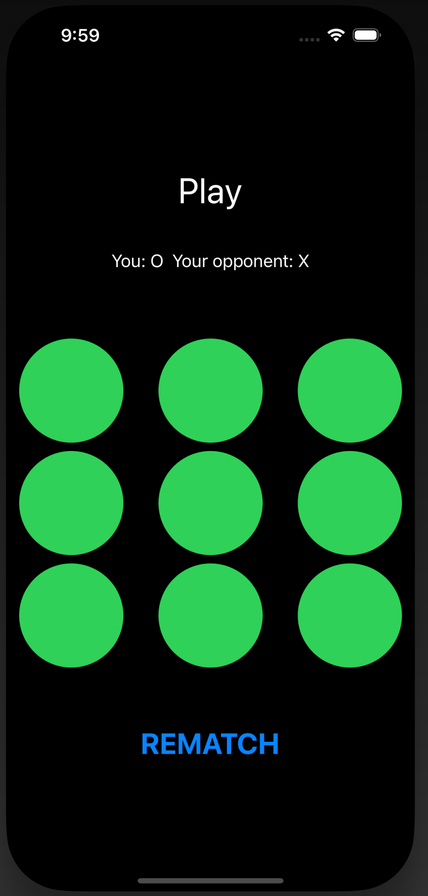
```
enum ViewState {
    case addPlayer
    case findingPlayers
    case play
}
```



.addPlayer

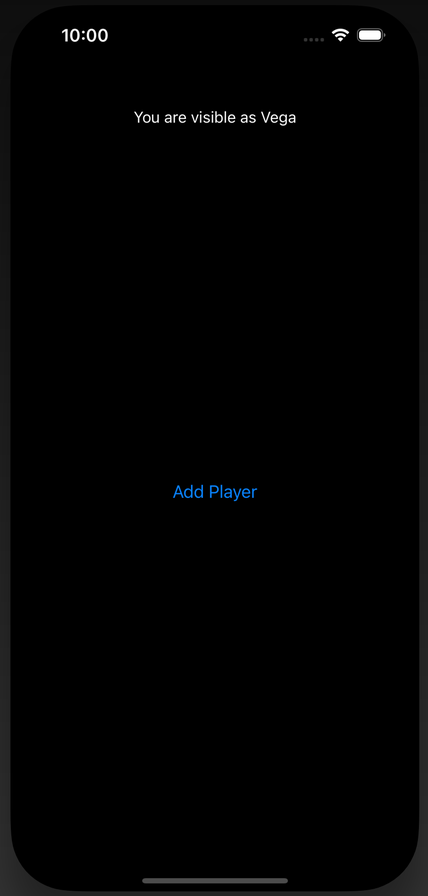


.findingPlayers



.play

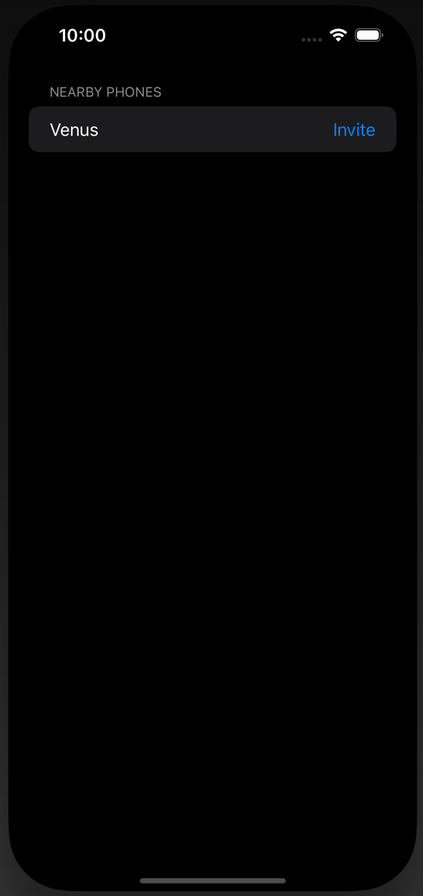
Add Player



.addPlayer

```
Button("Add Player") {
    mcManager.viewState = .findingPlayers
    // The beginner will be X
    mcManager.currentPlayer.playerType = .x
}
```

We need to have a list of all nearby peers



.findingPlayers

MCNearbyServiceBrowserDelegate

Informs us when a peer is found

Informs us when a peer is lost

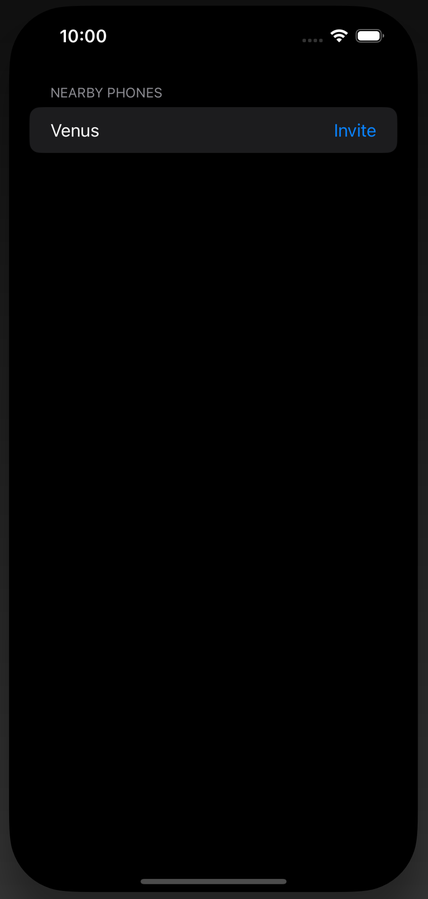
```
extension MCManager: MCNearbyServiceBrowserDelegate {
    func browser(_ browser: MCNearbyServiceBrowser, foundPeer peerID: MCPeerID,
                 withDiscoveryInfo info: [String : String]?) {
        guard availablePlayers.first(where: { $0.id == peerID }) == nil else {
            return
        }
        availablePlayers.update(with: Player(id: peerID, name: peerID.displayName))

    }

    func browser(_ browser: MCNearbyServiceBrowser, lostPeer peerID: MCPeerID) {
        guard let lostPeer = availablePlayers.first(where: {$0.id == peerID}) else { return }
        availablePlayers.remove(lostPeer)
    }

}
```

Showing Nearby Peers

Using availablePlayers list to show nearby peers

Inviting peers to join our session

```
List {
    Section {
        ForEach(mcManager.availablePlayers.sorted {
            $0.id.displayName < $1.id.displayName
        } , id: \.self) { peer in
            HStack {
                VStack(alignment: .leading) {
                    Text(peer.id.displayName)
                        .font(.body)
                }
                Spacer()
                switch peer.state {
                case .notConnected:
                    Button("Invite") {
                        mcManager.browser.invitePeer(peer.id, to: mcManager.session!,
                            withContext: nil, timeout: 30)

                    }
                    .foregroundColor(.blue)
                case .connected:
                    Image(uiImage: .checkmark)
                        .renderingMode(.template)
                        .foregroundColor(.green)

                case .connecting:
                    ProgressView()
```



MCNearbyServiceAdvertiserDelegate

Incoming invitation request

Automatically connect

Or show and alert for connection request

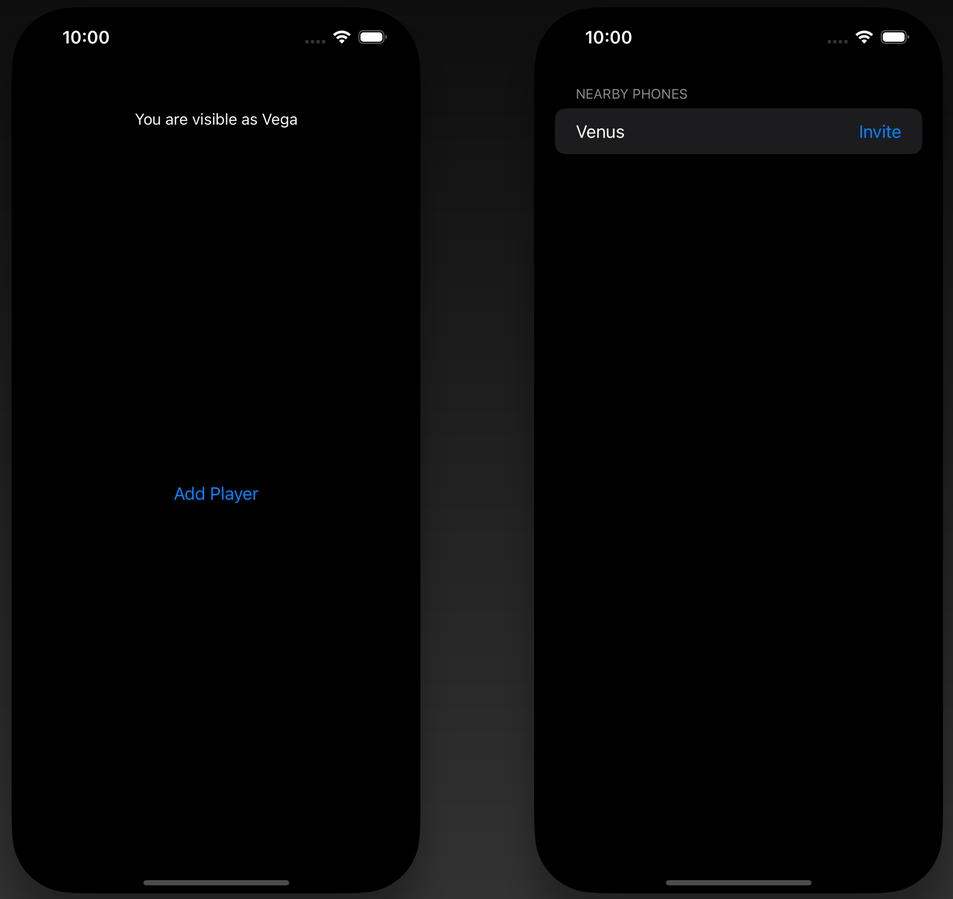
```
extension MCManager: MCNearbyServiceAdvertiserDelegate {

    func advertiser(_ advertiser: MCNearbyServiceAdvertiser, didReceiveInvitationFromPeer peerID:
        MCPeerID, withContext context: Data?, invitationHandler: @escaping (Bool, MCSession?) -> Void) {
        log.info("didReceiveInvitationFromPeer \(peerID) and we connect right away")
        invitationHandler(true, session)

    }

}
```

MCSessionDelegate

Called when the state of a nearby peer changes

```
extension MCManager: MCSessionDelegate {
    func session(_ session: MCSession, peer peerID: MCPeerID, didChange state: MCSessionState) {
        switch state {
        case .notConnected:
            DispatchQueue.main.async {•••}

        case .connected:
            DispatchQueue.main.async {•••}

        case .connecting:
            DispatchQueue.main.async {•••}
        default:
            log.info("\(peerID.displayName) unknown MCSessionState")
            break
        }
    }
```

When Connected



From .addPlayer or .findingPlayers

To .play

MCSessionDelegate

Data Handling - Sending

```
func send(Data, toPeers: [MCPeerID], with: MCSessionSendDataMode) throws
```

Sends a message to nearby peers.

```
func sendResource(at: URL, withName: String, toPeer: MCPeerID, withCompletionHandler:
(((any Error)?) -> Void)?) -> Progress?
```

Sends the contents of a URL to a peer.

```
func startStream(withName: String, toPeer: MCPeerID) throws -> OutputStream
```

Opens a byte stream to a nearby peer.

https://developer.apple.com/documentation/multipeerconnectivity/mcsession

Send playerType on Connection

The starter is X and the other is O

```
func sendPlayerTypeToOpponent(peerID: MCPeerID) {
    do {
        let data = try JSONEncoder().encode(Message(messageType: .playerType, playerType: .o,
            board: []) )
        try session?.send(data, toPeers: [peerID], with: .reliable)
        var peerToUpdate = self.availablePlayers.first(where: {
            $0.id == peerID
        })
        peerToUpdate?.playerType = .o
        // From here on we only work with opponent when sending messages
        self.opponent = peerToUpdate
        self.viewState = .play
    } catch {
        log.error("error failed sending playerType to opponent \(error.localizedDescription)")
    }
}
```

MCSessionDelegate

Data Handling - Message

```
enum MessageType: String, Codable {
    case playerType
    case move
}
```

```
struct Message: Codable {
    let messageType: MessageType
    let playerType: PlayerType?
    var board: [PlayerType?]
    var activePlayer: PlayerType?
    var winner: PlayerType?
}
```

MCSessionDelegate

Data Handling - Receiving

```
func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {
    if let message = try? JSONDecoder().decode(Message.self, from: data) {•••}
}
```

```
func session(_ session: MCSession, didStartReceivingResourceWithName resourceName:
    String, fromPeer peerID: MCPeerID, with progress: Progress) {}

func session(_ session: MCSession, didFinishReceivingResourceWithName resourceName:
    String, fromPeer peerID: MCPeerID, at localURL: URL?, withError error: Error?) {}
```

```
func session(_ session: MCSession, didReceive stream: InputStream, withName streamName:
    String, fromPeer peerID: MCPeerID) {}
```

MCSessionDelegate

Data Handling - Decoding

```
func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {

    if let message = try? JSONDecoder().decode(Message.self, from: data) {
        switch message.messageType {
        case .playerType:
            self.currentPlayer.playerType = message.playerType
            self.opponent = Player(id: peerID, name: peerID.displayName)
            self.opponent?.playerType = .x
            self.viewState = .play

        case .move:
            // TODO: Complete this for the game
            break
        }

    }

}
```

Game Logic

```
// For the game
var board:[PlayerType?] = Array(repeating: nil, count: 9)
var activePlayer: PlayerType = .x
var winner: PlayerType? = nil
```

```
@Observable
class TicTacToeViewModel {

    var mcManager = MCManager.share

    func buttonTapped(i: Int) {
        guard mcManager.currentPlayer.playerType == mcManager.activePlayer else {
            // It is not the turn of this player
            return
        }

        guard mcManager.board[i] == nil && mcManager.winner == nil else {
            return
        }
        mcManager.board[i] = mcManager.activePlayer

        if checkWinner() {
            mcManager.winner = mcManager.activePlayer
        } else {
            mcManager.activePlayer = mcManager.activePlayer == .x ? .o : .x
        }
        mcManager.sendGameState()
    }
```

Sending Game Logic

```
func sendGameState() {
    do {
        guard let opponent else {
            return
        }
```

```
func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {

    if let message = try? JSONDecoder().decode(Message.self, from: data) {
        switch message.messageType {
        case .playerType:
            ...
        case .move:
            self.board = message.board
            if let activePlayer = message.activePlayer {
                self.activePlayer = activePlayer
            }
            self.winner = message.winner

        }

    }

}
```

Demo Time!

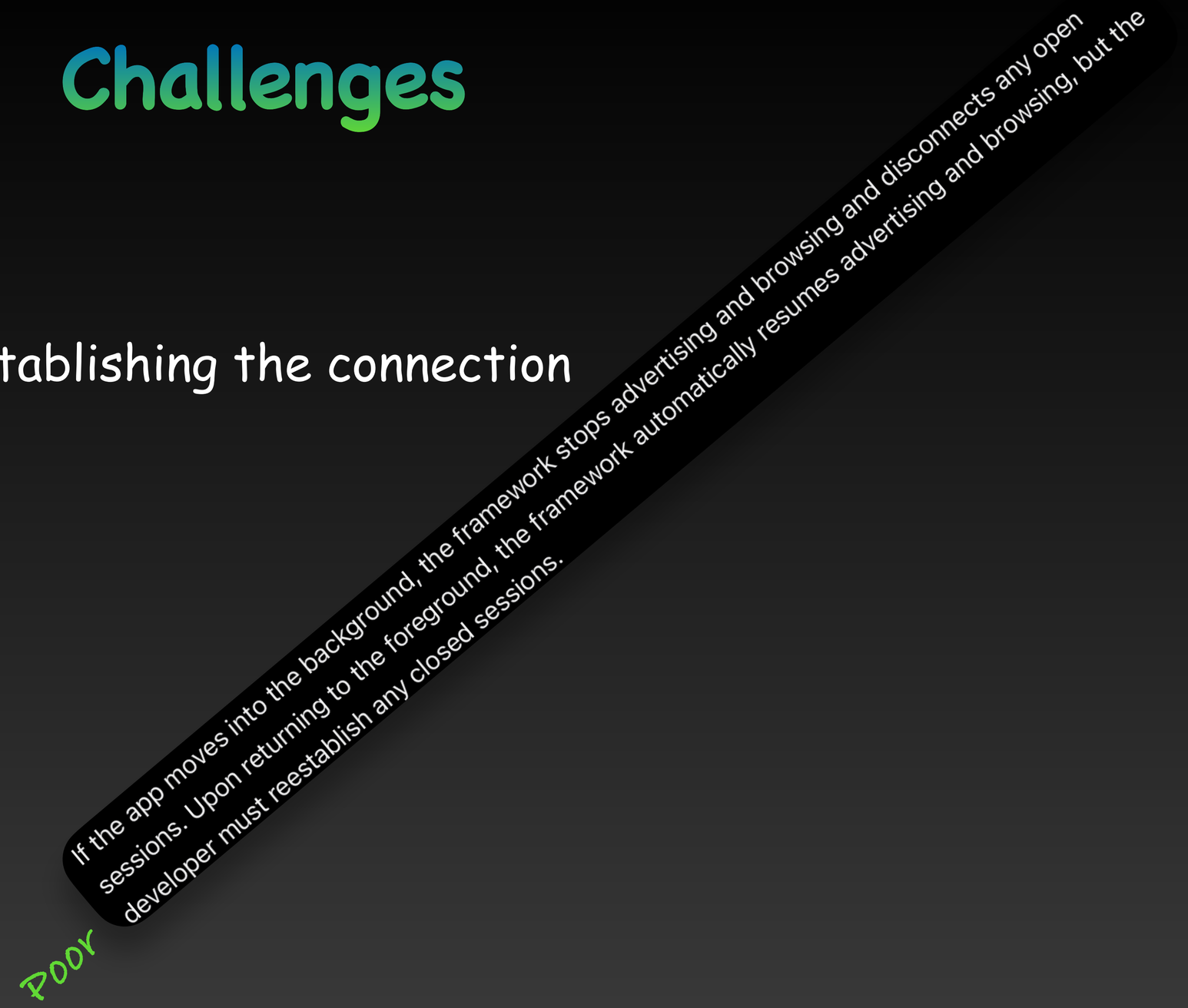
Let's play

Challenges

Background mode and reestablishing the connection

Debugging

Unexpected behaviour



Future of Multipeer Connectivity

Cross-Platform Compatibility

Enhanced Performance and Reliability

Integration with IoT and Smart Devices

Better support for background mode

Thank You

@zamzampooya
#WomenLifeFreedom