Distributed actors
и где они обитают

Eugene Antropov
iOS Expert Raiffeisen

О чем поговорим?

- Что такое акторы?
- А что такое «distributed actors»?
- Свой rest distributed actor?
- А что предлагает apple?
- MultiPeer distributed actor

Что такое актор?

```swift
class FormatterCache {
    private var formatters = [String: DateFormatter]()
    private let queue = DispatchQueue(label: "com.dw.FormatterCache.\(UUID().uuidString)")

    func formatter(for format: String) -> DateFormatter {
        return queue.sync {
            if let formatter = formatters[format] {
                return formatter
            }

            let formatter = DateFormatter()
            formatter.dateFormat = format
            formatters[format] = formatter

            return formatter
        }
    }
}
```

Что такое актор?

```swift
actor FormatterCache {
    private var formatters = [String: DateFormatter]()

    func formatter(for format: String) -> DateFormatter {
        if let formatter = formatters[format] {
            return formatter
        }
        let formatter = DateFormatter()
        formatter.dateFormat = format
        formatters[format] = formatter
        return formatter
    }
}

func usage() async {
    let formatter = await cache.formatter(for: "DD/MM/YYYY")
}
```

«Обычные» акторы

Александр Андрюхин

iOS Expert Raiffeisen

https://www.youtube.com/watch?v=b7G866hEbAY

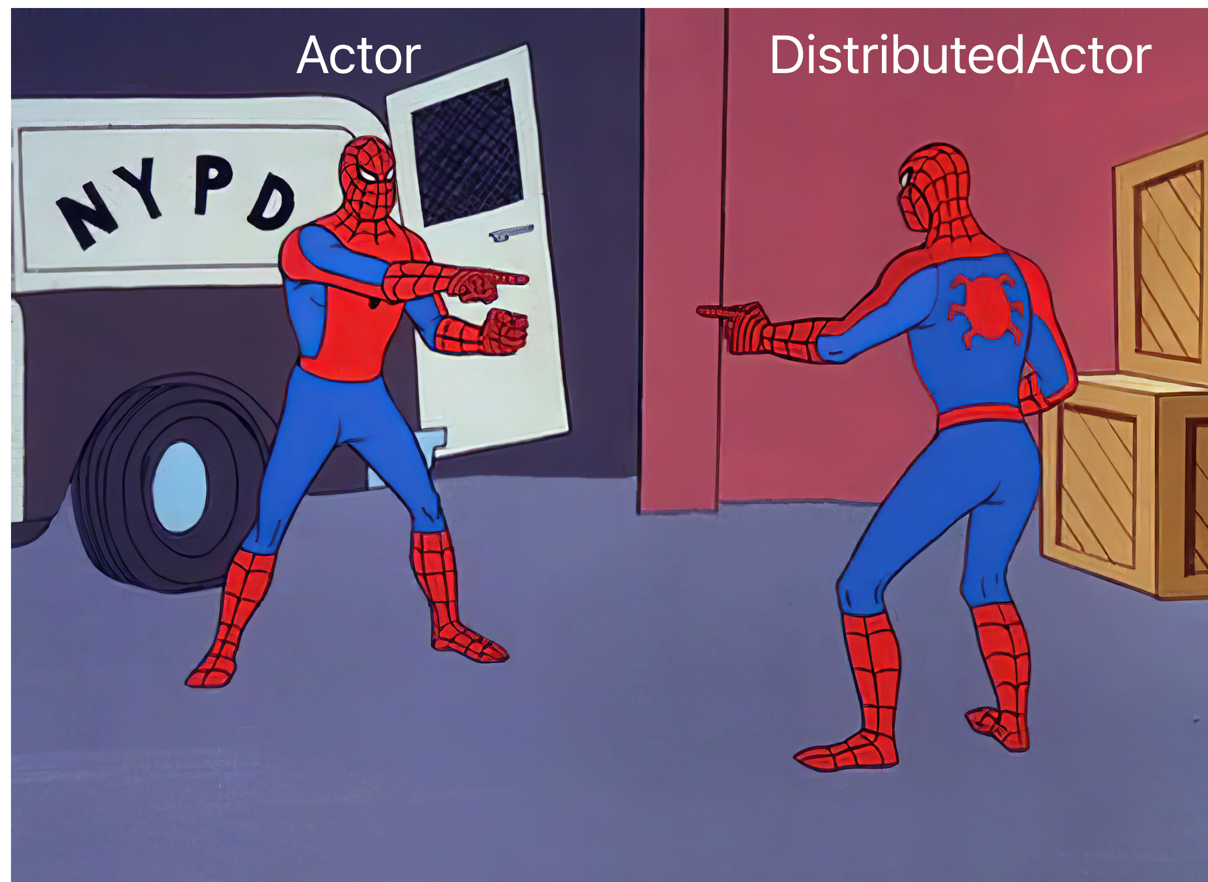
«Distributed» акторы

Что это такое?

«Distributed» акторы

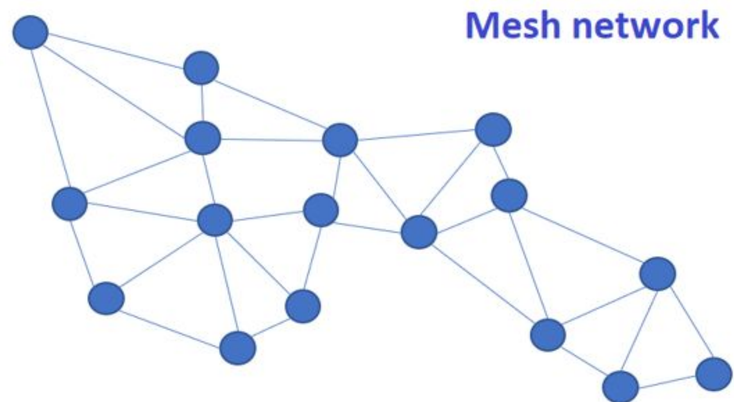
Что это такое?

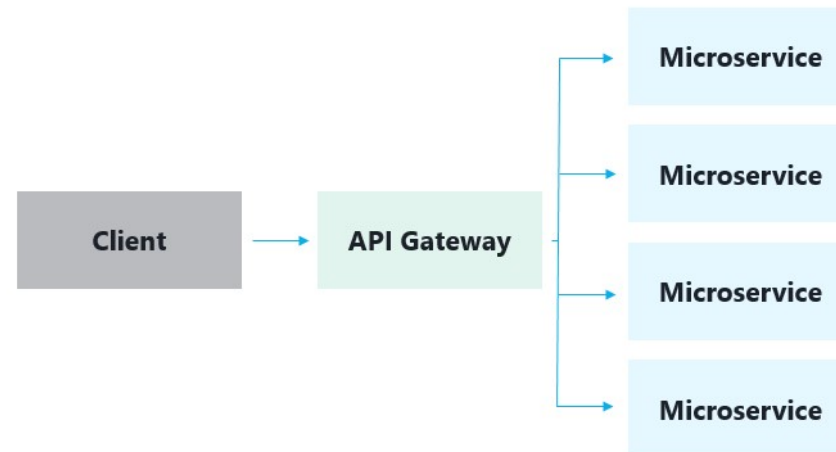
Да, тоже самое, только удаленные.

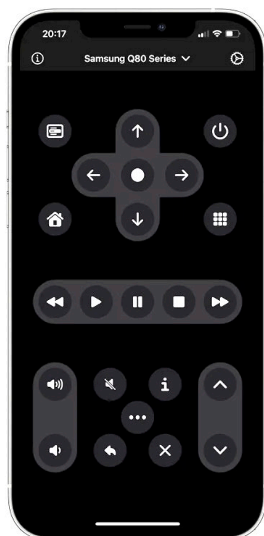


Конец

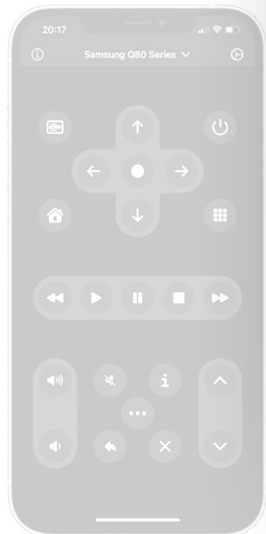
Для чего?







Для чего?




```swift
func startNewGame() async {
    let nearbyUsers = actorSystem.allNearbyUsers()
    let newGame = NewGame(actorSystem: actorSystem)

    for user in currentUser + nearbyUsers {
        user.startGame(newGame)
    }

    await newGame.startRound()
}


    …


distributed func startGame(_ game: NewGame) {
    …
}
```

Но все-же, давайте дадим им шанс

```
let user = actorSystem.resolve(User.self, id: 1)
try await user.repos.first?.issues.first?.remove()
```

Actor system

```swift
class RestActorSystem: Decodable {
    let baseURL: URL

    ...

    func resolve<T: RestPathable & Decodable>(_ type: T.Type, id: String) async throws {
        let request = URLRequest(url: baseURL.appending(path: T.path(for: id)))
        let data = try await URLSession(configuration: .default).data(for: request)
        let decoder = JSONDecoder()
        return try decoder.decode(type, from: data.0)
    }
}

let actorSystem = RestActorSystem(baseURL: URL(string: "https://github.com/api"))
```

Наш «Distributed» actor

```
protocol RestPathable {
    static func path(for id: String) -> String
}

actor User: RestPathable, Decodable {
    var id: String
    var reposID: [String]
    static func path(for id: String) -> String {
        return "/users/\(id)"
    }

    var repos: [Repo] {
        get async {
            var repos = [Repo]()
            for repo in reposID {
                if let repoObject = try? await actorSystem.resolve(Repo.self, id: repo) {
                    repos.append(repoObject)
                }
            }
            return repos
        }
    }
}
```

Наш «Distributed» actor

```
protocol RestPathable {
    static func path(for id: String) -> String
}

actor User: RestPathable, Decodable {
    var id: String
    var reposID: [String]
    static func path(for id: String) -> String {
        return "/users/\(id)"
    }

    var repos: [Repo] {
        get async {
            var repos = [Repo]()
            for repo in reposID {
                if let repoObject = try? await actorSystem.resolve(Repo.self, id: repo) {
                    repos.append(repoObject)
                }
            }
            return repos
        }
    }
}
```

Но? Зачем вообще «distributed» акторы

Но? Зачем вообще «distributed» акторы

На самом деле мы все сделали не правильно

```
let user = actorSystem.resolve(User.self, id: 1)
try await user.repos.first?.issues.first?.remove()
```

Наш «Distributed» actor

```
protocol RestPathable {
    static func path(for id: String) -> String
}

actor User: RestPathable, Decodable {
    var id: String
    var reposID: [String]
    static func path(for id: String) -> String {
        return "/users/\(id)"
    }

    var repos: [Repo] {
        get async {
            var repos = [Repo]()
            for repo in reposID {
                if let repoObject = try? await actorSystem.resolve(Repo.self, id: repo) {
                    repos.append(repoObject)
                }
            }
            return repos
        }
    }
}
```

А что сделал apple

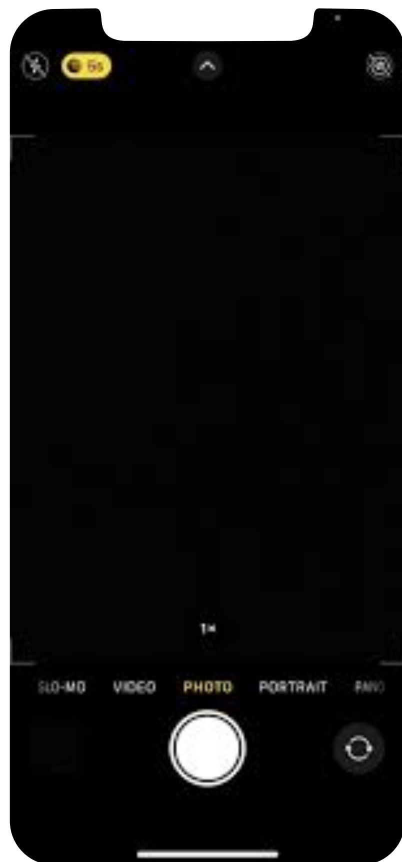
```
import Distributed

@available(macOS 13.0, iOS 16.0, watchOS 9.0, tvOS 16.0, *)
public protocol DistributedActor

@available(macOS 13.0, iOS 16.0, watchOS 9.0, tvOS 16.0, *)
public protocol DistributedActorSystem
```

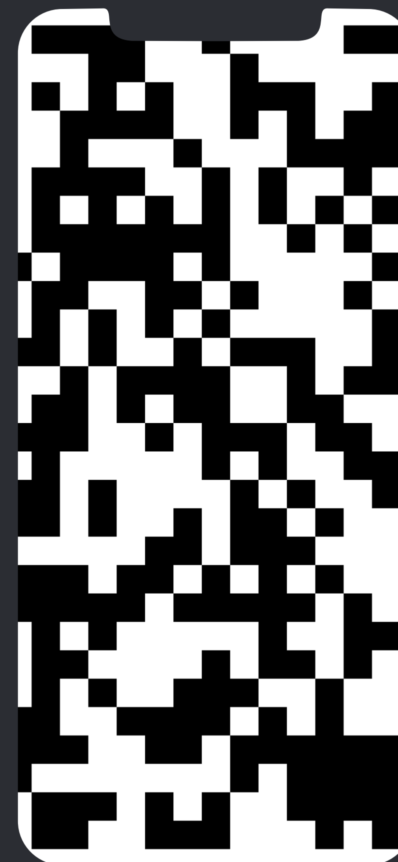
https://github.com/apple/swift-distributed-actors

Как это работает?

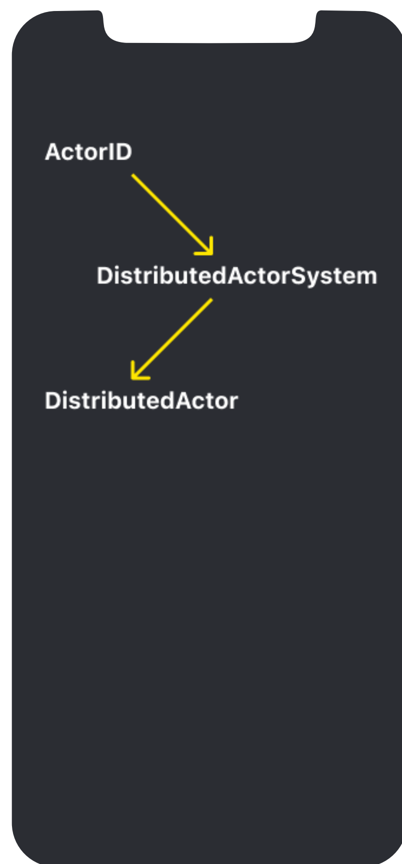


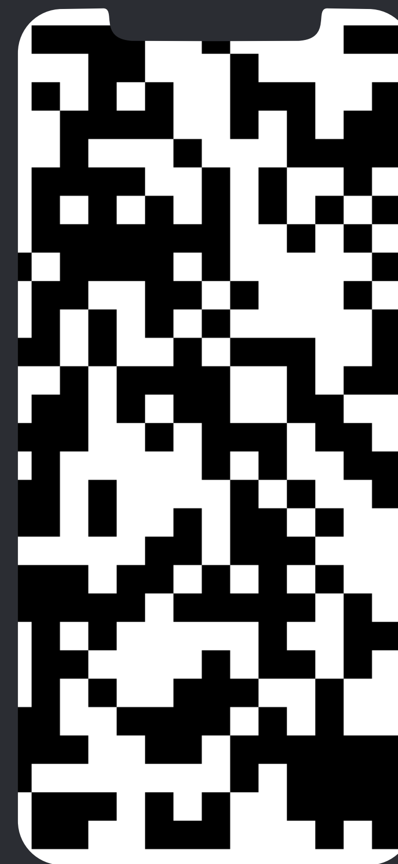
ActorID

DistributedActorSystem

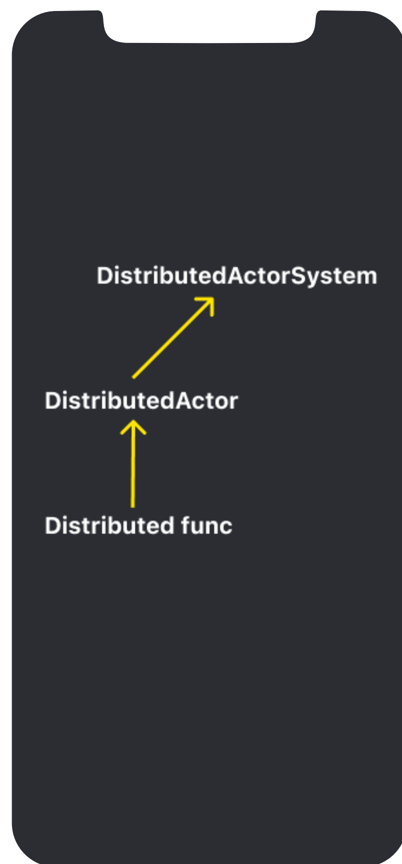


Как это работает?

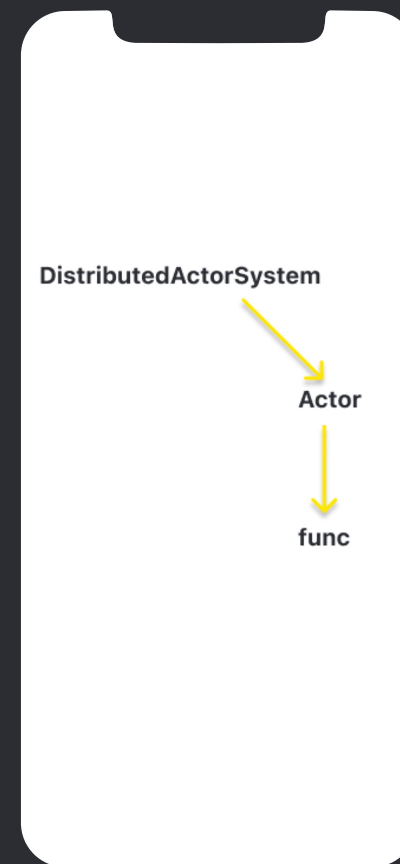




Как это работает?



Request



Что нужно что-бы это заработало

DistributedActorSystem

```
associatedtype ActorID : Hashable, Sendable
```

DistributedActorSystem

```
associatedtype ActorID : Hashable, Sendable
```

```
associatedtype SerializationRequirement
```

DistributedActorSystem

```
associatedtype ActorID : Hashable, Sendable
```

```
associatedtype SerializationRequirement
```

```
associatedtype InvocationEncoder : DistributedTargetInvocationEncoder
```

DistributedActorSystem

```
associatedtype ActorID : Hashable, Sendable
```

```
associatedtype SerializationRequirement
```

```
associatedtype InvocationEncoder : DistributedTargetInvocationEncoder
```

```
associatedtype InvocationDecoder : DistributedTargetInvocationDecoder
```

DistributedActorSystem

```
associatedtype ActorID : Hashable, Sendable
```

```
associatedtype SerializationRequirement
```

```
associatedtype InvocationEncoder : DistributedTargetInvocationEncoder
```

```
associatedtype InvocationDecoder : DistributedTargetInvocationDecoder
```

```
associatedtype ResultHandler : DistributedTargetInvocationResultHandler
```

DistributedActor

```swift
associatedtype ActorSystem : DistributedActorSystem
```

```swift
typealias DefaultDistributedActorSystem = SomeSystem
```

DistributedActor

```
distributed actor MyDistributedActor: DistributedActor {
   distributed func myFunc() {

      print("Distributed and remote device")

      whenLocal {
         print("Only on local device")
      }

   }
}
```

Попробуем сделать

Попробуем сделать

MultiPeer Connectivity.Framework

```
public final class MultipeerActorSystem: DistributedActorSystem, @unchecked Sendable {
    public typealias ActorID = ActorIdentity
    public typealias InvocationEncoder = NearbyActorSystemCallEncoder
    public typealias InvocationDecoder = NearbyActorSystemCallDecoder
    public typealias ResultHandler = NearbyActorSystemResultHandler<MultipeerTransport>
    public typealias SerializationRequirement = any Codable
    ...
```

ActorIdentity

```swift
public typealias HostIdentity = UUID
public typealias LocalIdentity = UUID


public struct ActorIdentity: Hashable, Sendable, Codable {
    public let hostID: HostIdentity
    public let localID: LocalIdentity

    public init(hostID: HostIdentity, localID: LocalIdentity) {
        self.hostID = hostID
        self.localID = localID
    }

    public var id: String {
        return "\(hostID)-\(localID)"
    }
}
```

ActorIdentity

```swift
public typealias HostIdentity = UUID
public typealias LocalIdentity = UUID


public struct ActorIdentity: Hashable, Sendable, Codable {
    public let hostID: HostIdentity
    public let localID: LocalIdentity

    public init(hostID: HostIdentity, localID: LocalIdentity) {
        self.hostID = hostID
        self.localID = localID
    }

    public var id: String {
        return "\(hostID)-\(localID)"
    }
}
```

ActorIdentity

```swift
public typealias HostIdentity = UUID
public typealias LocalIdentity = UUID


public struct ActorIdentity: Hashable, Sendable, Codable {
    public let hostID: HostIdentity
    public let localID: LocalIdentity

    public init(hostID: HostIdentity, localID: LocalIdentity) {
        self.hostID = hostID
        self.localID = localID
    }

    public var id: String {
        return "\(hostID)-\(localID)"
    }
}
```

InvocationEncoder

```
public class MultipeerActorSystemCallEncoder: DistributedTargetInvocationEncoder,
    @unchecked Sendable {
    public typealias SerializationRequirement = any Codable

    var genericSubs: [String] = []
    var argumentData: [Data] = []

    public func recordGenericSubstitution<T>(_ type: T.Type) throws {
        if let name = _mangledTypeName(T.self) {
            genericSubs.append(name)
        }
    }

    public func recordArgument<Value: Codable>(_ argument: RemoteCallArgument<Value>) throws {
        let data = try JSONEncoder().encode(argument.value)
        self.argumentData.append(data)
    }

    public func recordReturnType<R: Codable>(_ type: R.Type) throws { }

    public func recordErrorType<E: Error>(_ type: E.Type) throws { }

    public func doneRecording() throws { }
}
```

InvocationEncoder

```swift
public class MultipeerActorSystemCallEncoder: DistributedTargetInvocationEncoder,
    @unchecked Sendable {
    public typealias SerializationRequirement = any Codable

    var genericSubs: [String] = []
    var argumentData: [Data] = []

    public func recordGenericSubstitution<T>(_ type: T.Type) throws {
        if let name = _mangledTypeName(T.self) {
            genericSubs.append(name)
        }
    }

    public func recordArgument<Value: Codable>(_ argument: RemoteCallArgument<Value>) throws {
        let data = try JSONEncoder().encode(argument.value)
        self.argumentData.append(data)
    }

    public func recordReturnType<R: Codable>(_ type: R.Type) throws { }

    public func recordErrorType<E: Error>(_ type: E.Type) throws { }

    public func doneRecording() throws { }
}
```

InvocationEncoder

```
public class MultipeerActorSystemCallEncoder: DistributedTargetInvocationEncoder,
    @unchecked Sendable {
    public typealias SerializationRequirement = any Codable

    var genericSubs: [String] = []
    var argumentData: [Data] = []

    public func recordGenericSubstitution<T>(_ type: T.Type) throws {
        if let name = _mangledTypeName(T.self) {
            genericSubs.append(name)
        }
    }

    public func recordArgument<Value: Codable>(_ argument: RemoteCallArgument<Value>) throws {
        let data = try JSONEncoder().encode(argument.value)
        self.argumentData.append(data)
    }

    public func recordReturnType<R: Codable>(_ type: R.Type) throws { }

    public func recordErrorType<E: Error>(_ type: E.Type) throws { }

    public func doneRecording() throws { }
}
```

InvocationEncoder

```
public class MultipeerActorSystemCallEncoder: DistributedTargetInvocationEncoder,
    @unchecked Sendable {
    public typealias SerializationRequirement = any Codable

    var genericSubs: [String] = []
    var argumentData: [Data] = []

    public func recordGenericSubstitution<T>(_ type: T.Type) throws {
        if let name = _mangledTypeName(T.self) {
            genericSubs.append(name)
        }
    }

    public func recordArgument<Value: Codable>(_ argument: RemoteCallArgument<Value>) throws {
        let data = try JSONEncoder().encode(argument.value)
        self.argumentData.append(data)
    }

    public func recordReturnType<R: Codable>(_ type: R.Type) throws { }

    public func recordErrorType<E: Error>(_ type: E.Type) throws { }

    public func doneRecording() throws { }
}
```

InvocationDecoder

```
public class MultipeerActorSystemCallDecoder: DistributedTargetInvocationDecoder {
    public typealias SerializationRequirement = Codable

    let decoder = JSONDecoder()
    let envelope: RemoteCallEnvelope
    var argumentsIterator: Array<Data>.Iterator

    init(system: any DistributedActorSystem, envelope: RemoteCallEnvelope) {
        self.envelope = envelope
        self.argumentsIterator = envelope.args.makeIterator()
        decoder.userInfo[.actorSystemKey] = system
    }

    public  func decodeGenericSubstitutions() throws -> [Any.Type] {
        envelope.genericSubs.compactMap { name in
            _typeByName(name)
        }
    }

    public  func decodeNextArgument<Argument: Codable>() throws -> Argument {
        let data = argumentsIterator.next()
        return try decoder.decode(Argument.self, from: data)
    }

    public func decodeErrorType() throws -> Any.Type? {
        nil
    }

    public func decodeReturnType() throws -> Any.Type? {
        nil
    }
}
```

InvocationDecoder

```
public class MultipeerActorSystemCallDecoder: DistributedTargetInvocationDecoder {
    public typealias SerializationRequirement = Codable

    let decoder = JSONDecoder()
    let envelope: RemoteCallEnvelope
    var argumentsIterator: Array<Data>.Iterator

    init(system: any DistributedActorSystem, envelope: RemoteCallEnvelope) {
        self.envelope = envelope
        self.argumentsIterator = envelope.args.makeIterator()
        decoder.userInfo[.actorSystemKey] = system
    }

    public  func decodeGenericSubstitutions() throws -> [Any.Type] {
        envelope.genericSubs.compactMap { name in
            _typeByName(name)
        }
    }

    public  func decodeNextArgument<Argument: Codable>() throws -> Argument {
        let data = argumentsIterator.next()
        return try decoder.decode(Argument.self, from: data)
    }

    public func decodeErrorType() throws -> Any.Type? {
        nil
    }

    public func decodeReturnType() throws -> Any.Type? {
        nil
    }
}
```

InvocationDecoder

```swift
public class MultipeerActorSystemCallDecoder: DistributedTargetInvocationDecoder {
    public typealias SerializationRequirement = Codable

    let decoder = JSONDecoder()
    let envelope: RemoteCallEnvelope
    var argumentsIterator: Array<Data>.Iterator

    init(system: any DistributedActorSystem, envelope: RemoteCallEnvelope) {
        self.envelope = envelope
        self.argumentsIterator = envelope.args.makeIterator()
        decoder.userInfo[.actorSystemKey] = system
    }

    public  func decodeGenericSubstitutions() throws -> [Any.Type] {
        envelope.genericSubs.compactMap { name in
            _typeByName(name)
        }
    }

    public  func decodeNextArgument<Argument: Codable>() throws -> Argument {
        let data = argumentsIterator.next()
        return try decoder.decode(Argument.self, from: data)
    }

    public func decodeErrorType() throws -> Any.Type? {
        nil
    }

    public func decodeReturnType() throws -> Any.Type? {
        nil
    }
}
```

InvocationDecoder

```
public class MultipeerActorSystemCallDecoder: DistributedTargetInvocationDecoder {
    public typealias SerializationRequirement = Codable

    let decoder = JSONDecoder()
    let envelope: RemoteCallEnvelope
    var argumentsIterator: Array<Data>.Iterator

    init(system: any DistributedActorSystem, envelope: RemoteCallEnvelope) {
        self.envelope = envelope
        self.argumentsIterator = envelope.args.makeIterator()
        decoder.userInfo[.actorSystemKey] = system
    }

    public  func decodeGenericSubstitutions() throws -> [Any.Type] {
        envelope.genericSubs.compactMap { name in
            _typeByName(name)
        }
    }

    public  func decodeNextArgument<Argument: Codable>() throws -> Argument {
        let data = argumentsIterator.next()
        return try decoder.decode(Argument.self, from: data)
    }

    public func decodeErrorType() throws -> Any.Type? {
        nil
    }

    public func decodeReturnType() throws -> Any.Type? {
        nil
    }
}
```

Result handler

```swift
public struct MultipeerActorSystemResultHandler: DistributedTargetInvocationResultHandler {
    public typealias SerializationRequirement = Codable

    let callID: CallID
    let caller: HostIdentity
    let callee: ActorIdentity
    let transport: Tranport

    public func onReturn<Success: SerializationRequirement>(value: Success) async throws {
        let encoder = JSONEncoder()
        let returnValue = try encoder.encode(value)
        let envelope = ReplyEnvelope(callID: callID, caller: caller, callee: callee, value: returnValue)
        try await transport.sendRemoteReplay(envelope: envelope)
    }

    public func onReturnVoid() async throws {
        let envelope = ReplyEnvelope(callID: callID, caller: caller, callee: callee)
        try await transport.sendRemoteReplay(envelope: envelope)
    }

    public func onThrow<Err: Error>(error: Err) async throws {
        let envelope = ReplyEnvelope(callID: callID, caller: caller, callee: callee, error: error)
        try await transport.sendRemoteReplay(envelope: envelope)
    }
}
```

Result handler

```swift
public struct MultipeerActorSystemResultHandler: DistributedTargetInvocationResultHandler {
    public typealias SerializationRequirement = Codable

    let callID: CallID
    let caller: HostIdentity
    let callee: ActorIdentity
    let transport: Tranport

    public func onReturn<Success: SerializationRequirement>(value: Success) async throws {
        let encoder = JSONEncoder()
        let returnValue = try encoder.encode(value)
        let envelope = ReplyEnvelope(callID: callID, caller: caller, callee: callee, value: returnValue)
        try await transport.sendRemoteReplay(envelope: envelope)
    }

    public func onReturnVoid() async throws {
        let envelope = ReplyEnvelope(callID: callID, caller: caller, callee: callee)
        try await transport.sendRemoteReplay(envelope: envelope)
    }

    public func onThrow<Err: Error>(error: Err) async throws {
        let envelope = ReplyEnvelope(callID: callID, caller: caller, callee: callee, error: error)
        try await transport.sendRemoteReplay(envelope: envelope)
    }
}
```

Result handler

```swift
public struct MultipeerActorSystemResultHandler: DistributedTargetInvocationResultHandler {
    public typealias SerializationRequirement = Codable

    let callID: CallID
    let caller: HostIdentity
    let callee: ActorIdentity
    let transport: Tranport

    public func onReturn<Success: SerializationRequirement>(value: Success) async throws {
        let encoder = JSONEncoder()
        let returnValue = try encoder.encode(value)
        let envelope = ReplyEnvelope(callID: callID, caller: caller, callee: callee, value: returnValue)
        try await transport.sendRemoteReplay(envelope: envelope)
    }

    public func onReturnVoid() async throws {
        let envelope = ReplyEnvelope(callID: callID, caller: caller, callee: callee)
        try await transport.sendRemoteReplay(envelope: envelope)
    }

    public func onThrow<Err: Error>(error: Err) async throws {
        let envelope = ReplyEnvelope(callID: callID, caller: caller, callee: callee, error: error)
        try await transport.sendRemoteReplay(envelope: envelope)
    }
}
```

Ну и на этом....

Ну и на этом....

Самое просто позади

DistributedActorSystem

```
public func resolve<Act>(id: ActorIdentity, as actorType: Act.Type) throws -> Act? where Act : DistributedActor, ActorIdentity == Act.ID {
    <#code#>
}

public func assignID<Act>(_ actorType: Act.Type) -> ActorIdentity where Act : DistributedActor, ActorIdentity == Act.ID {
    <#code#>
}

public func actorReady<Act>(_ actor: Act) where Act : DistributedActor, ActorIdentity == Act.ID {
    <#code#>
}

public func resignID(_ id: ActorIdentity) {
    <#code#>
}

public func makeInvocationEncoder() -> NearbyActorSystemCallEncoder {
    <#code#>
}
```

DistributedActorSystem

```swift
var knownActors = [ActorIdentity: WeakBoxActor]()
public func resolve<Act>(id: ActorIdentity, as actorType: Act.Type) throws -> Act? where Act : DistributedActor, ActorIdentity == Act.ID {
    return knownActors[id] as? Act
}
```

```swift
    public func assignID<Act>(_ actorType: Act.Type) -> ActorIdentity where Act : DistributedActor, ActorIdentity == Act.ID {
// Создали актор через .init(actorSystem: DistributedActorSystem)
    }

    public func actorReady<Act>(_ actor: Act) where Act : DistributedActor, ActorIdentity == Act.ID {
// Сохраняем id готового актора
    }

    public func resignID(_ id: ActorIdentity) {
// Освобождаем id на deinit
    }
```

DistributedActorSystem

```
var knownActors = [ActorIdentity: WeakBoxActor]()
public func resolve<Act>(id: ActorIdentity, as actorType: Act.Type) throws -> Act? where Act : DistributedActor, ActorIdentity == Act.ID {
    return knownActors[id] as? Act
}
```

```
    public func assignID<Act>(_ actorType: Act.Type) -> ActorIdentity where Act : DistributedActor, ActorIdentity == Act.ID {
// Создали актор через .init(actorSystem: DistributedActorSystem)
    }

    public func actorReady<Act>(_ actor: Act) where Act : DistributedActor, ActorIdentity == Act.ID {
// Сохраняем id готового актора
    }

    public func resignID(_ id: ActorIdentity) {
// Освобождаем id на deinit
    }
```

DistributedActorSystem

```
var knownActors = [ActorIdentity: WeakBoxActor]()
public func resolve<Act>(id: ActorIdentity, as actorType: Act.Type) throws -> Act? where Act : DistributedActor, ActorIdentity == Act.ID {
    return knownActors[id] as? Act
}
```

```
    public func assignID<Act>(_ actorType: Act.Type) -> ActorIdentity where Act : DistributedActor, ActorIdentity == Act.ID {
// Создали актор через .init(actorSystem: DistributedActorSystem)
    }

    public func actorReady<Act>(_ actor: Act) where Act : DistributedActor, ActorIdentity == Act.ID {
// Сохраняем id готового актора
    }

    public func resignID(_ id: ActorIdentity) {
// Освобождаем id на deinit
    }
```

DistributedActorSystem

```
var knownActors = [ActorIdentity: WeakBoxActor]()
public func resolve<Act>(id: ActorIdentity, as actorType: Act.Type) throws -> Act? where Act : DistributedActor, ActorIdentity == Act.ID {
    return knownActors[id] as? Act
}
```

```
    public func assignID<Act>(_ actorType: Act.Type) -> ActorIdentity where Act : DistributedActor, ActorIdentity == Act.ID {
// Создали актор через .init(actorSystem: DistributedActorSystem)
    }

    public func actorReady<Act>(_ actor: Act) where Act : DistributedActor, ActorIdentity == Act.ID {
// Сохраняем id готового актора
    }

    public func resignID(_ id: ActorIdentity) {
// Освобождаем id на deinit
    }
```

DistributedActorSystem

```swift
public func makeInvocationEncoder() -> NearbyActorSystemCallEncoder {
    .init()
}
```

MultipeerActorSystem

```
public func resolve<Act>(id: ActorIdentity, as actorType: Act.Type) throws -> Act? where Act : DistributedActor, ActorIdentity == Act.ID {
    <#code#>
}

public func assignID<Act>(_ actorType: Act.Type) -> ActorIdentity where Act : DistributedActor, ActorIdentity == Act.ID {
    <#code#>
}

public func actorReady<Act>(_ actor: Act) where Act : DistributedActor, ActorIdentity == Act.ID {
    <#code#>
}

public func resignID(_ id: ActorIdentity) {
    <#code#>
}

public func makeInvocationEncoder() -> NearbyActorSystemCallEncoder {
    <#code#>
}
```

DistributedActorSystem

```
public func resolve<Act>(id: ActorIdentity, as actorType: Act.Type) throws -> Act? where Act : DistributedActor, ActorIdentity == Act.ID {
    <#code#>
}
```

Class 'MultipeerActorSystem' is missing witness for protocol requirement 'remoteCall'

Class 'MultipeerActorSystem' is missing witness for protocol requirement 'remoteCallVoid'

```
    <#code#>
}
```

MultipeerActorSystem

	Initial Declaration	Extension
Value Type	Static	Static
Class	Table (Virtual)	Static
Protocol	Table (Witness)	Static
NSObject Subclass	Table (Virtual)	Message

DistributedActorSystem

Class 'Multipeer ocol requirement
'remoteCall'

Class 'Multipeer ocol requirement
'remoteCallVoid

```
func remoteCall<Act, Err, Res>(
    on actor: Act,
    target: RemoteCallTarget,
    invocation: inout InvocationEncoder,
    throwing: Err.Type,
    returning: Res.Type
) async throws -> Res
  where Act: DistributedActor,
        Act.ID == ActorID,
        Err: Error,
        Res: Codable {

        }

func remoteCallVoid<Act, Err>(
    on actor: Act,
    target: RemoteCallTarget,
    invocation: inout InvocationEncoder,
    throwing: Err.Type
) async throws
  where Act: DistributedActor,
        Act.ID == ActorID,
        Err: Error {

        }
```

remoteCall

```
func remoteCall<Act, Err, Res>(
    on actor: Act,
    target: RemoteCallTarget,
    invocation: inout InvocationEncoder,
    throwing: Err.Type,
    returning: Res.Type
) async throws -> Res
where Act: DistributedActor,
      Act.ID == ActorIdentity,
      Err: Error,
      Res: Codable {
          let callEnvelope = RemoteCallEnvelope(
            caller: transport.currentIdentity,
            callee: actor.id,
            callID: UUID(),
            invocationTarget: target.identifier,
            genericSubs: invocation.genericSubs,
            args: invocation.argumentData
          )
          let reply = try await transport.sendRemoteCall(envelope: callEnvelope)

          let decoder = JSONDecoder()
          decoder.userInfo[.actorSystemKey] = self

          do {
              return try decoder.decode(Res.self, from: reply.value)
          } catch {
              throw NearbyActorSystemError.failedDecodingResponse(data: replyData, error: error)
          }
      }
```

remoteCall

```
func remoteCall<Act, Err, Res>(
    on actor: Act,
    target: RemoteCallTarget,
    invocation: inout InvocationEncoder,
    throwing: Err.Type,
    returning: Res.Type
) async throws -> Res
where Act: DistributedActor,
      Act.ID == ActorIdentity,
      Err: Error,
      Res: Codable {
          let callEnvelope = RemoteCallEnvelope(
            caller: transport.currentIdentity,
            callee: actor.id,
            callID: UUID(),
            invocationTarget: target.identifier,
            genericSubs: invocation.genericSubs,
            args: invocation.argumentData
          )
          let reply = try await transport.sendRemoteCall(envelope: callEnvelope)

          let decoder = JSONDecoder()
          decoder.userInfo[.actorSystemKey] = self

          do {
              return try decoder.decode(Res.self, from: reply.value)
          } catch {
              throw NearbyActorSystemError.failedDecodingResponse(data: replyData, error: error)
          }
      }
```

remoteCall

```
func remoteCall<Act, Err, Res>(
    on actor: Act,
    target: RemoteCallTarget,
    invocation: inout InvocationEncoder,
    throwing: Err.Type,
    returning: Res.Type
) async throws -> Res
where Act: DistributedActor,
      Act.ID == ActorIdentity,
      Err: Error,
      Res: Codable {
          let callEnvelope = RemoteCallEnvelope(
            caller: transport.currentIdentity,
            callee: actor.id,
            callID: UUID(),
            invocationTarget: target.identifier,
            genericSubs: invocation.genericSubs,
            args: invocation.argumentData
          )
          let reply = try await transport.sendRemoteCall(envelope: callEnvelope)

          let decoder = JSONDecoder()
          decoder.userInfo[.actorSystemKey] = self

          do {
              return try decoder.decode(Res.self, from: reply.value)
          } catch {
              throw NearbyActorSystemError.failedDecodingResponse(data: replyData, error: error)
          }
      }
```

remoteCall

```
func remoteCall<Act, Err, Res>(
    on actor: Act,
    target: RemoteCallTarget,
    invocation: inout InvocationEncoder,
    throwing: Err.Type,
    returning: Res.Type
) async throws -> Res
where Act: DistributedActor,
      Act.ID == ActorIdentity,
      Err: Error,
      Res: Codable {
          let callEnvelope = RemoteCallEnvelope(
            caller: transport.currentIdentity,
            callee: actor.id,
            callID: UUID(),
            invocationTarget: target.identifier,
            genericSubs: invocation.genericSubs,
            args: invocation.argumentData
          )
          let reply = try await transport.sendRemoteCall(envelope: callEnvelope)

          let decoder = JSONDecoder()
          decoder.userInfo[.actorSystemKey] = self

          do {
              return try decoder.decode(Res.self, from: reply.value)
          } catch {
              throw NearbyActorSystemError.failedDecodingResponse(data: replyData, error: error)
          }
      }
```

handleInbound

```swift
func handleInbound() {
        Task {
            for await envelope in self.transport.inboundEnveloperHandler() {
                guard let recipient = knowActors[id]?.value else {
                    // Unknown actor, better to replay caller
                    return
                }
                let target = RemoteCallTarget(envelope.invocationTarget)
                var decoder = Self.InvocationDecoder(system: self, envelope: envelope)
                let handler = Self.ResultHandler(
                    callID: envelope.callID,
                    caller: envelope.caller,
                    callee: envelope.callee,
                    transport: transport
                )
                do {
                    try await executeDistributedTarget(
                        on: recipient,
                        target: target,
                        invocationDecoder: &decoder,
                        handler: handler
                    )
                } catch let error as (Codable & Error) {
                    try! await handler.onThrow(error: error)
                }
            }
        }
    }
```

handleInbound

```
func handleInbound() {
        Task {
            for await envelope in self.transport.inboundEnveloperHandler() {
                guard let recipient = knowActors[id]?.value else {
                    // Unknown actor, better to replay caller
                    return
                }
                let target = RemoteCallTarget(envelope.invocationTarget)
                var decoder = Self.InvocationDecoder(system: self, envelope: envelope)
                let handler = Self.ResultHandler(
                    callID: envelope.callID,
                    caller: envelope.caller,
                    callee: envelope.callee,
                    transport: transport
                )
                do {
                    try await executeDistributedTarget(
                        on: recipient,
                        target: target,
                        invocationDecoder: &decoder,
                        handler: handler
                    )
                } catch let error as (Codable & Error) {
                    try! await handler.onThrow(error: error)
                }
            }
        }
    }
```

handleInbound

```
func handleInbound() {
        Task {
            for await envelope in self.transport.inboundEnveloperHandler() {
                guard let recipient = knowActors[id]?.value else {
                    // Unknown actor, better to replay caller
                    return
                }
                let target = RemoteCallTarget(envelope.invocationTarget)
                var decoder = Self.InvocationDecoder(system: self, envelope: envelope)
                let handler = Self.ResultHandler(
                    callID: envelope.callID,
                    caller: envelope.caller,
                    callee: envelope.callee,
                    transport: transport
                )
                do {
                    try await executeDistributedTarget(
                        on: recipient,
                        target: target,
                        invocationDecoder: &decoder,
                        handler: handler
                    )
                } catch let error as (Codable & Error) {
                    try! await handler.onThrow(error: error)
                }
            }
        }
    }
```

handleInbound

```
func handleInbound() {
        Task {
            for await envelope in self.transport.inboundEnveloperHandler() {
                guard let recipient = knowActors[id]?.value else {
                    // Unknown actor, better to replay caller
                    return
                }
                let target = RemoteCallTarget(envelope.invocationTarget)
                var decoder = Self.InvocationDecoder(system: self, envelope: envelope)
                let handler = Self.ResultHandler(
                    callID: envelope.callID,
                    caller: envelope.caller,
                    callee: envelope.callee,
                    transport: transport
                )
                do {
                    try await executeDistributedTarget(
                        on: recipient,
                        target: target,
                        invocationDecoder: &decoder,
                        handler: handler
                    )
                } catch let error as (Codable & Error) {
                    try! await handler.onThrow(error: error)
                }
            }
        }
    }
```

Вот теперь все

Вот теперь все

С actorSystem, а теперь чуть-чуть транспорт

Transport

```swift
var inFlightRequest = InFlightRequestsHolder()

public func sendRemoteCall(envelope: RemoteCallEnvelope) async throws -> ReplyEnvelope {
    try await withCheckedThrowingContinuation { continuation in
        Task {
            await inFlightRequest.addRequest(id: envelope.callID, continuation: continuation)
            let peer = try await self.peerForHost(for: envelope.callee.hostID)
            do {
                let session = await sessionProvider.sessionFor(peer: peer)
                let jsonEncoder = JSONEncoder()
                let data = try jsonEncoder.encode(object)
                try session.send(data, toPeers: [peer], with: .reliable)
            } catch {
                await inFlightRequest.handleError(id: envelope.callID, error: .sendError)
            }
        }
    }
}
```

Transport

```swift
var inFlightRequest = InFlightRequestsHolder()

public func sendRemoteCall(envelope: RemoteCallEnvelope) async throws -> ReplyEnvelope {
    try await withCheckedThrowingContinuation { continuation in
        Task {
            await inFlightRequest.addRequest(id: envelope.callID, continuation: continuation)
            let peer = try await self.peerForHost(for: envelope.callee.hostID)
            do {
                let session = await sessionProvider.sessionFor(peer: peer)
                let jsonEncoder = JSONEncoder()
                let data = try jsonEncoder.encode(object)
                try session.send(data, toPeers: [peer], with: .reliable)
            } catch {
                await inFlightRequest.handleError(id: envelope.callID, error: .sendError)
            }
        }
    }
}
```

Transport

```
var inFlightRequest = InFlightRequestsHolder()

public func sendRemoteCall(envelope: RemoteCallEnvelope) async throws -> ReplyEnvelope {
    try await withCheckedThrowingContinuation { continuation in
        Task {
            await inFlightRequest.addRequest(id: envelope.callID, continuation: continuation)
            let peer = try await self.peerForHost(for: envelope.callee.hostID)
            do {
                let session = await sessionProvider.sessionFor(peer: peer)
                let jsonEncoder = JSONEncoder()
                let data = try jsonEncoder.encode(object)
                try session.send(data, toPeers: [peer], with: .reliable)
            } catch {
                await inFlightRequest.handleError(id: envelope.callID, error: .sendError)
            }
        }
    }
}
```

Transport

```swift
var inFlightRequest = InFlightRequestsHolder()

public func sendRemoteCall(envelope: RemoteCallEnvelope) async throws -> ReplyEnvelope {
    try await withCheckedThrowingContinuation { continuation in
        Task {
            await inFlightRequest.addRequest(id: envelope.callID, continuation: continuation)
            let peer = try await self.peerForHost(for: envelope.callee.hostID)
            do {
                let session = await sessionProvider.sessionFor(peer: peer)
                let jsonEncoder = JSONEncoder()
                let data = try jsonEncoder.encode(object)
                try session.send(data, toPeers: [peer], with: .reliable)
            } catch {
                await inFlightRequest.handleError(id: envelope.callID, error: .sendError)
            }
        }
    }
}
```

Transport

```
var inFlightRequest = InFlightRequestsHolder()

public func sendRemoteCall(envelope: RemoteCallEnvelope) async throws -> ReplyEnvelope {
    try await withCheckedThrowingContinuation { continuation in
        Task {
            await inFlightRequest.addRequest(id: envelope.callID, continuation: continuation)
            let peer = try await self.peerForHost(for: envelope.callee.hostID)
            do {
                let session = await sessionProvider.sessionFor(peer: peer)
                let jsonEncoder = JSONEncoder()
                let data = try jsonEncoder.encode(object)
                try session.send(data, toPeers: [peer], with: .reliable)
            } catch {
                await inFlightRequest.handleError(id: envelope.callID, error: .sendError)
            }
        }
    }
}
```

Transport

```
var inFlightRequest = InFlightRequestsHolder()

public func sendRemoteCall(envelope: RemoteCallEnvelope) async throws -> ReplyEnvelope {
    try await withCheckedThrowingContinuation { continuation in
        Task {
            await inFlightRequest.addRequest(id: envelope.callID, continuation: continuation)
            let peer = try await self.peerForHost(for: envelope.callee.hostID)
            do {
                let session = await sessionProvider.sessionFor(peer: peer)
                let jsonEncoder = JSONEncoder()
                let data = try jsonEncoder.encode(object)
                try session.send(data, toPeers: [peer], with: .reliable)
            } catch {
                await inFlightRequest.handleError(id: envelope.callID, error: .sendError)
            }
        }
    }
}
```

didReceive data

```
public func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {
    if let remoteCall = try? jsonDecoder.decode(RemoteCallEnvelope.self, from: data) {
      if remoteCall.callee.hostID == currentIdentity {
            inboundEnveloperSubject.send(remoteCall)
        }
      }
      if let remoteReplay = try? jsonDecoder.decode(ReplyEnvelope.self, from: data) {
          Task {
              if remoteReplay.caller == currentIdentity {
                  await inFlightRequest.handleResponse(id: remoteReplay.callID, response: remoteReplay)
              }
          }
          return
      }
}
```

didReceive data

```
public func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {
   if let remoteCall = try? jsonDecoder.decode(RemoteCallEnvelope.self, from: data) {
    if remoteCall.callee.hostID == currentIdentity {
            inboundEnveloperSubject.send(remoteCall)
        }
    }
    if let remoteReplay = try? jsonDecoder.decode(ReplyEnvelope.self, from: data) {
        Task {
            if remoteReplay.caller == currentIdentity {
                await inFlightRequest.handleResponse(id: remoteReplay.callID, response: remoteReplay)
            }
        }
        return
    }
}
```

didReceive data

```
public func session(_ session: MCSession, didReceive data: Data, fromPeer peerID: MCPeerID) {
  if let remoteCall = try? jsonDecoder.decode(RemoteCallEnvelope.self, from: data) {
    if remoteCall.callee.hostID == currentIdentity {
            inboundEnveloperSubject.send(remoteCall)
        }
    }
    if let remoteReplay = try? jsonDecoder.decode(ReplyEnvelope.self, from: data) {
        Task {
            if remoteReplay.caller == currentIdentity {
                await inFlightRequest.handleResponse(id: remoteReplay.callID, response: remoteReplay)
            }
        }
        return
    }
}
```

Вот теперь точно все

Спасибо

https://github.com/antigp/NearbyActorSystem.git

Eugene Antropov
iOS Expert Raiffeisen

https://stackoverflow.com/questions/74140663/increase-the-duration-of-the-floating-thumbnail-screenshot-preview-on-macos/74142493