Game Kit Programming Guide

Networking & Internet

2009-05-28

Contents

Figures

Introduction

The Game Kit framework provides features designed for game developers who want to connect users of different iPhones together. Game Kit includes the following technologies:

- **Peer-to-peer connectivity** allows your application to create an ad-hoc Bluetooth network between multiple iPhones. Although designed with games in mind, this network is useful for any type of data exchange among users of your application. For example, an application could use peer-to-peer connectivity to share electronic business cards or other data.
- **In-game voice** allows your application to provide voice communication between two iPhones. In Game Voice uses your application to create its own network connection between the two users.

Who Should Read This Document

You should read this document if you want your application to connect the user's iPhone to other local devices over Bluetooth, or if your application wants to include voice chat.

Organization of This Document

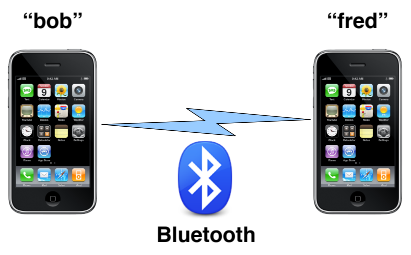
This document contains the following articles:

- "Peer-to-Peer Connectivity" (page 9) provides an overview of the connectivity features included in the Game Kit framework.
- "Finding Peers with Peer Picker" (page 15) shows how an application can use a peer picker to allow an iPhone user to connect to a copy of your application running on another user's iPhone.
- "Working with Sessions" (page 17) explains how your application uses a session configured by the Peer Picker.
- "In-Game Voice" (page 19) provides an overview of the voice technologies available in the Game Kit framework.
- "Adding Voice Chat" (page 23) explains how to add voice communication to your application using a session to connect the two iPhones.

Peer-to-Peer Connectivity

The `GKSession` class allows your application to create and manage an ad-hoc Bluetooth network, as shown in Figure 1. Copies of your application running on multiple devices can discover each other and exchange information, providing a simple and powerful way to create multiplayer games on the iPhone. Further, sessions offer all applications an exciting new way to allow users to collaborate with each other.

Figure 1 Bluetooth networking



Bluetooth networking is not supported on the original iPhone or the first-generation iPod Touch. It is also not supported in Simulator.

When you develop a peer-to-peer application, you can either implement your own user interface to show other users discovered by the session or you can use a `GKPeerPickerController` object to present a standard user interface to configure a session between two iPhones.

Once the network between the devices is established, the `GKSession` class does not dictate format for the data transmitted over it. You are free to design data formats that are optimal for your application.

Note: This guide discusses the infrastructure provided by the peer-to-peer connectivity classes. It does not cover the design and implementation of networked games or applications.

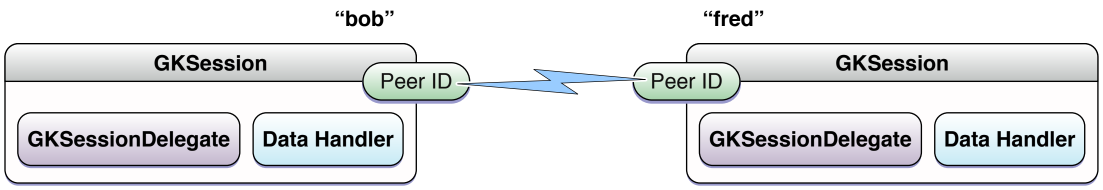
Sessions

Sessions are created, they discover each other, and they are connected into a network. Your application uses the connected session to transmit data to other iPhones. Your application provides a delegate to handle connection requests and a data handler to receive data directed to your application from another iPhone.

Peers

iPhones connected to the ad-hoc network are known as **peers**. A peer is synonymous with a session running inside your application. Each session creates a unique **peer identification** string, or `peerID`, used to identify them to other users on the network. Interactions with other peers on the network are done through their peer ID. For example, if your application knows the peer ID of another peer, it can retrieve a user-readable name for that peer by calling the session's `displayNameForPeer:` method, as shown in Figure 2

Figure 2 Peer IDs are used to interact with other peers



Other peers on the network can appear in a variety of states relative to the local session. Peers can appear or disappear from the network, be connected to the session, or disconnect from the session. Your application implements the delegate's `session:peer:didChangeState:` method to be notified when peers change their state.

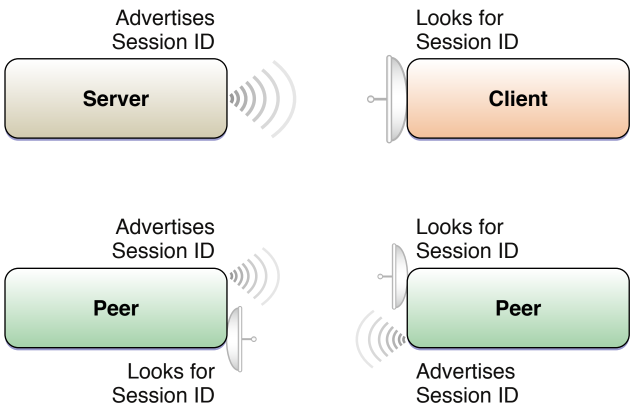
Discovering Other Peers

Every session implements its own specific type of service. This might be a specific game or a feature like swapping business cards. You are responsible for determining the needs of your service type and the data it needs to exchange between peers.

Sessions discover other peers on the network based on a **session mode** which is set when the session is initialized. Your application can configure the session to be a **server**, which advertises a service type on the network; a **client**, which searches for advertising servers; or a **peer**, which advertises like a server and searches like a client simultaneously. Figure 3 illustrates the session mode.

Servers advertise their service type with a **session identification** string, or `sessionID`. Clients find only servers with a matching session ID.

Figure 3 Servers, clients, and peers



The session ID is the short name of a registered Bonjour service. For more information on Bonjour services, see Bonjour Networking. If you do not specify a session ID when creating a session, the session generates one using the application's bundle identifier.

To establish a connection, at least one iPhone must advertise as a server and another must search for it. Your application provides code for both modes. Peers, which advertise and search simultaneously, are the most flexible way to implement this. However, because they both advertise and search, it takes longer for other devices to be detected by the session.

Implementing a Server

A copy of your application acting as a server initializes the session by calling `initWithSessionID:displayName:sessionMode:` with a session mode of either `GKSessionModeServer` or `GKSessionModePeer`. After the application configures the session, it advertises the service by setting the session's `isAvailable` property to `YES`.

Servers are notified when a client requests a connection. When the client sends a connection request, the `session:didReceiveConnectionRequestFromPeer:` method on the delegate is called. A typical behavior of the delegate should be to use the `peerID` string to retrieve a user-readable name by calling `displayNameForPeer:`. It can then present an interface that lets users decide whether to accept the connection.

The delegate accepts the request by calling the session's `acceptConnectionFromPeer:error:` or rejects it by calling `denyConnectionFromPeer:`.

When the connection is successfully created, the delegate's `session:peer:didChangeState:` method is called to inform the delegate that a new peer is connected.

Connecting to a Service

A copy of your application acting as a client initializes the session by calling `initWithSessionID:displayName:sessionMode:` with a session mode of either `GKSessionModeClient` or `GKSessionModePeer`. After configuring the session, your application searches the network for advertising servers by setting the session's `isAvailable` property to `YES`. If the session is configured with the `GKSessionModePeer` session mode it also advertises itself as a server, as described above.

When a client discovers an available server, the delegate's `session:peer:didChangeState:` method is called to provide the `peerID` string of the discovered server. Your application can call `displayNameForPeer:` to retrieve a user-readable name to display to the user. When the user selects a peer to connect to, your application calls the session's `connectToPeer:withTimeout:` method to request the connection.

When the connection is successfully created, the delegate's `session:peer:didChangeState:` method is called to inform the application that a new peer is connected.

Exchanging Data

Peers connected to the session can exchange data with other connected peers. Your application sends data to all connected peers by calling the `sendDataToAllPeers:withDataMode:error:` method or to a subset of the peers by calling the `sendData:toPeers:withDataMode:error:` method. The data is an arbitrary block of memory encapsulated in an `NSData` object. Your application can design and use any data formats it wishes for its data. Your application is free to create its own data format. For best performance, it is recommended that the size of the data objects be kept small (under 1000 bytes in length). Larger messages (up to 95 kilobytes) may need to be split into smaller chunks and reassembled at the destination, incurring additional latency and overhead.

You can choose to send data **reliably**, where the session retransmits data that fails to reach its destination, or **unreliably**, where it sends it only once. Unreliable messages are appropriate when the data must arrive in real time to be useful to other peers, and where sending an updated packet is more important than resending stale data (for example, dead reckoning information).

Reliable messages are received by participants in the order they were sent by the sender.

To receive data sent by other peers, your application implements the `receiveData:fromPeer:inSession:context:` method on an object. Your application provides this object to the session by calling the `setDataReceiveHandler:withContext:` method. When data is received from connected peers, the data handler is called on your application's main thread.

Important: All data received from other peers should be treated as *untrusted* data. Be sure to validate the data you receive from other peers and write your code carefully to avoid security vulnerabilities. See the *Secure Coding Guide* for more information.

Disconnecting Peers

When your application is ready to end a session, it should call the `disconnectFromAllPeers` method.

Your application can call the `disconnectPeerFromAllPeers:` method to disconnect a particular peer from the connection.

Networks are inherently unreliable. If a peer is non responsive for a period of time, it is automatically disconnected from the session. Your application can modify the `disconnectTimeout` property to control how long the session waits for another peer before disconnecting it.

Your application can detect when another peer disconnects inside the delegate's `session:peer:didChangeState:` method.

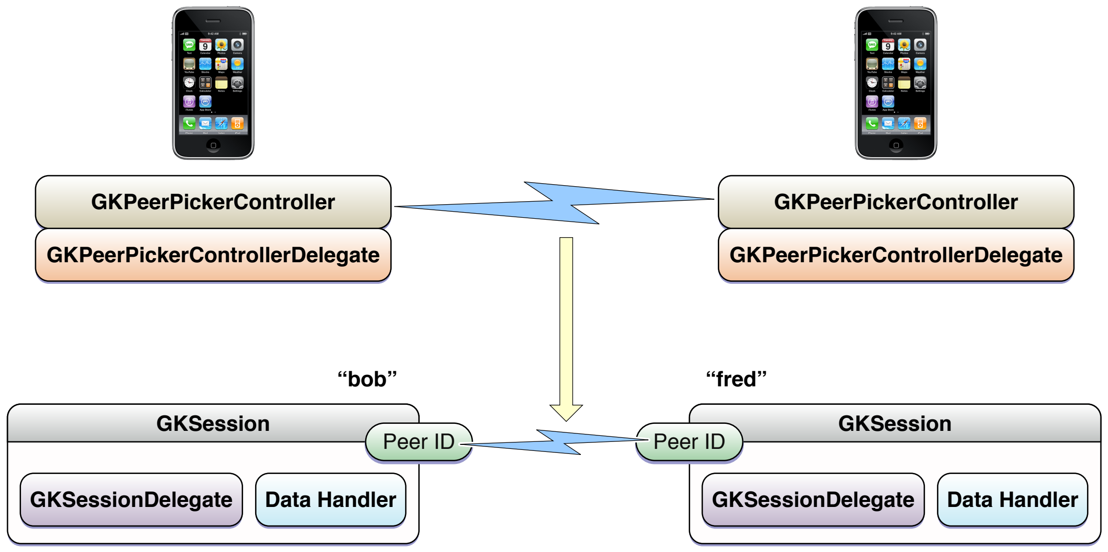
Cleaning Up

When your application is ready to dispose of the session, your application should disconnect from other peers, set the `isAvailable` flag to `NO`, remove the data handler and delegate, and then release the session.

The Peer Picker

While you may choose to implement your own user interface using the `GKSession`'s delegate, Game Kit offers a standard user interface to the discovery and connection process. A `GKPeerPickerController` object presents the user interface and responds to the user's actions, resulting in a fully configured `GKSession` that connects the two peers. Figure 4 illustrates how the peer picker works..

Figure 4 The peer picker creates a session connecting two peers on the network



Configuring the Peer Picker Controller

Your application provides a delegate that the controller calls as the user interacts with the peer picker.

The peer picker controller's `setConnectionTypesMask:` property is used to configure the list of available connection methods the application offers to the user. In iOS 3.0, the peer picker can select between local Bluetooth networking and Internet networking. When your application sets the mask to include more than one form of network, the peer picker controller displays an additional dialog to allow users to choose which network they want to use. When a user picks a network, the controller calls the delegate's `peerPickerController:didSelectConnectionType:` method.

Important: In iOS 3.0, the peer picker does not configure Internet connections. If your application provides Internet connections, when the user selects an Internet connection, your application must dismiss the peer picker and present its own user interface to configure the Internet connection.

If your application wants to customize the session created by the peer picker, it can implement the delegate's `peerPickerController:sessionForConnectionType:` method. If your application does not implement this method, the peer picker creates a default session for your application.

Displaying the Peer Picker

When your application has configured the peer picker controller, it shows the user interface by calling the controller's `show` method. If the user connects to another peer, the delegate's `peerPickerController:didConnectPeer:toSession:` method is called. Your application should take ownership of the session and call the controller's `dismiss` method to hide the dialog.

If the user cancels the connection attempt, the delegate's `peerPickerControllerDidCancel:` method is called.

Finding Peers with Peer Picker

The peer picker provides a standard user interface for connecting two users via Bluetooth. Optionally, your application can configure the peer picker to allow a user to choose between an Internet and Bluetooth connection. If an Internet connection is chosen, your application must dismiss the peer picker dialog and present its own user interface to complete the connection.

After you've read this article, you should read "Working with Sessions" (page 17) to see what your application can do with the created session.

To add a peer picker to your application, create a new class to hold the peer picker controller's delegate methods. Follow these steps:

1. Create and initialize a `GKPeerPickerController` object.

   ```
   picker = [[GKPeerPickerController alloc] init];
   ```

2. Attach the delegate (you'll define its methods as you proceed through these steps).

   ```
   picker.delegate = self;
   ```

3. Configure the allowed network types.

   ```
   picker.connectionTypesMask = GKPeerPickerConnectionTypeNearby |
   GKPeerPickerConnectionTypeOnline;
   ```

   Normally, the peer picker defaults to Bluetooth connections only. Your application may also add Internet (online) connections to the connection types mask. If your application does this, it must also implement the `peerPickerController:didSelectConnectionType:` method.

4. Optionally, implement the `peerPickerController:didSelectConnectionType:` method to dismiss the dialog when an Internet connection is selected.

   ```
   - (void)peerPickerController:(GKPeerPickerController *)picker
   didSelectConnectionType:(GKPeerPickerConnectionType)type {
       if (type == GKPeerPickerConnectionTypeOnline) {
           picker.delegate = nil;
           [picker dismiss];
           [picker autorelease];
          // Implement your own internet user interface here.
       }
   }
   ```

5. Implement the delegate's `peerPickerController:sessionForConnectionType:` method.

   ```
   - (GKSession *)peerPickerController:(GKPeerPickerController *)picker
   sessionForConnectionType:(GKPeerPickerConnectionType)type
   {
       GKSession* session = [[GKSession alloc] initWithSessionID:myExampleSessionID
    displayName:myName sessionMode:GKSessionModePeer];
       [session autorelease];
   ```

```
    return session;
}
```

Your application needs to implement this only if it wants to override the standard behavior of the peer picker controller.

6. Implement the delegate's `peerPickerController:didConnectPeer:toSession:` method to take ownership of the configured session.

```
- (void)peerPickerController:(GKPeerPickerController *)picker
didConnectPeer:(NSString *)peerID toSession: (GKSession *) session {
// Use a retaining property to take ownership of the session.
    self.gameSession = session;
// Assumes our object will also become the session's delegate.
    session.delegate = self;
    [session setDataReceiveHandler: self withContext:nil];
// Remove the picker.
    picker.delegate = nil;
    [picker dismiss];
    [picker autorelease];
// Start your game.
}
```

7. Your application also needs to implement the `peerPickerControllerDidCancel:` method to react when the user cancels the picker.

```
- (void)peerPickerControllerDidCancel:(GKPeerPickerController *)picker
{
    picker.delegate = nil;
    // The controller dismisses the dialog automatically.
    [picker autorelease];
}
```

8. Add code to show the dialog in your application.

```
[picker show];
```

Working with Sessions

This article explains how to use a `GKSession` object that was configured by the peer picker. For more information on how to configure the peer picker, see "Finding Peers with Peer Picker" (page 15).

A session receives two kinds of data: information about other peers, and data sent by connected peers. Your application provides a delegate to receive information about other peers and a data handler to receive information from other peers.

To use a session inside your application, first follow the steps found in "Finding Peers with Peer Picker" (page 15), then continue here.

1. Implement the session delegate's `session:peer:didChangeState:` method.

   The session's delegate is informed when another peer changes states relative to the session. Most of these states are handled automatically by the peer picker. If your application implements its own user interface, it should handle all state changes. For now the application should react when users connect and disconnect from the network.

   ```
   - (void)session:(GKSession *)session peer:(NSString *)peerID
   didChangeState:(GKPeerConnectionState)state
   {
       switch (state)
       {
           case GKPeerStateConnected:
   // Record the peerID of the other peer.
   // Inform your game that a peer has connected.
           break;
           case GKPeerStateDisconnected:
   // Inform your game that a peer has left.
           break;
       }
   }
   ```

2. Send data to other peers.

   ```
   - (void) mySendDataToPeers: (NSData *) data
   {
       [session sendDataToAllPeers: data withDataMode: GKSendDataReliable error:
   nil];
   }
   ```

3. Receive data from other peers.

   ```
   - (void) receiveData:(NSData *)data fromPeer:(NSString *)peer inSession:
   (GKSession *)session context:(void *)context
   {
       // Read the bytes in data and perform an application-specific action.
   }
   ```

Your application can either choose to process the data immediately, or retain it and process it later within your application. Your application should avoid lengthy computations within this method.

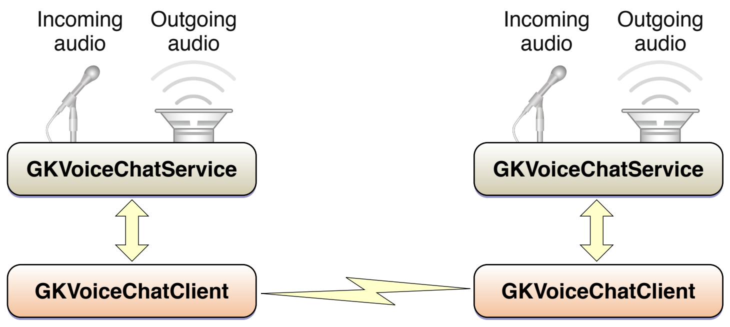
4. Clean up the session when you are ready to end the connection.

```
[session disconnectFromAllPeers];
session.available = NO;
[session setDataReceiveHandler: nil withContext: nil];
session.delegate = nil;
[session release];
```

In-Game Voice

A GKVoiceChatService object allows your application to easily create a voice chat between two iPhones, as shown in Figure 1. The voice chat service samples the microphone and plays audio received from the other participant. In-game voice relies on your application to provide a client that implements the GKVoiceChatClient protocol. The primary responsibility of the client is to connect the two participants together so that the voice chat service can exchange configuration data.

Figure 1 In Game Voice



Configuring a Voice Chat

Participant Identifiers

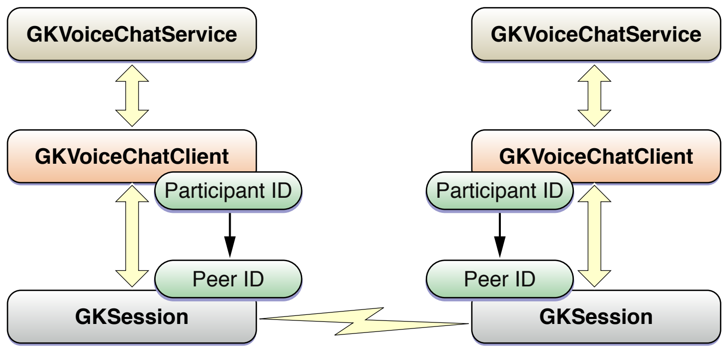
Each participant in a voice chat is identified by a unique **participant identifier** string provided by your client. The format and meaning of a participant identifier string is left to your client to decide.

Discovering other Participants

The voice chat service uses the client's network connection to exchange configuration data between the participants in order to create its direct connection between the two. However, the voice chat service does not provide a mechanism to discover the participant identifier of other participants. Your application is responsible for providing the participant identifiers of other users and translating these identifiers into connections to other participants.

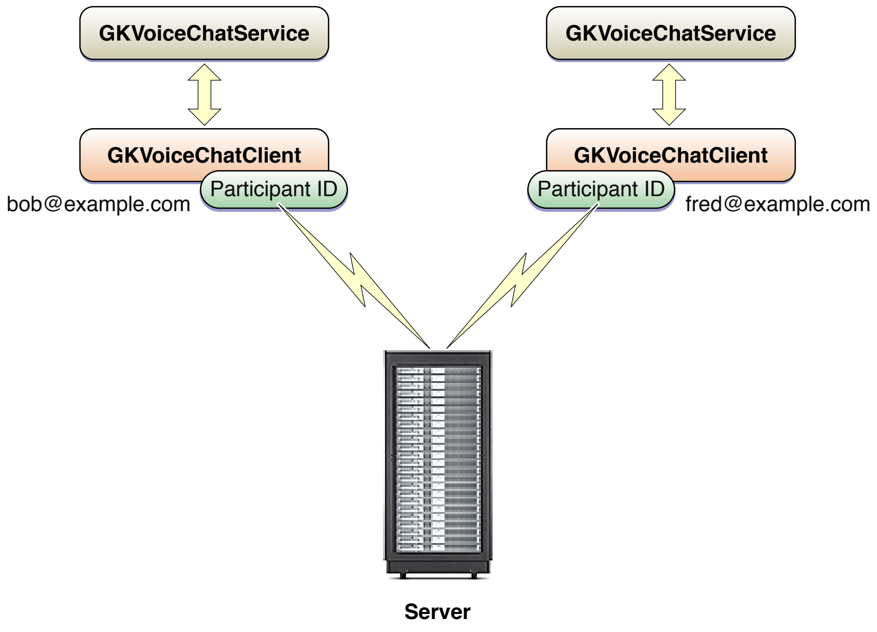
For example, if your application is already connected to another device through a GKSession object (see "Peer to Peer Connectivity" (page 9)), then each peer on the network is already uniquely identified by a peerID string. The session already knows the peerID string of the other participant. The client could reuse each peer's ID as the participant identifier and use the session to send and receive data, as shown in Figure 2

Figure 2 Peer-to-peer–based discovery



If the two devices are not directly aware of each other, your application needs another service to allow the two participants to discover each other and connect. In Figure 3, the server identifies participants with their email addresses and can route data between them.

Figure 3 Server-based discovery



Depending on the design of the server, it may either provide the list of participant identifiers to the clients or the user may need to provide the participant identifier (email address) of another user. In either case, the server is an intermediary that transmits data between the two users.

When the voice chat service wants to send its configuration data to another participant, it calls the client's `voiceChatService:sendData:toParticipantID:` method. The client must be able to reliably and promptly send the data to the other participant. When the other client receives the data, it forwards it to the service by calling the service's `receivedData:fromParticipantID:` method. The voice chat service uses this connection to configure its own real-time network connection between the two participants. The voice chat service uses the client's connection only to create its own connection.

Real-time Data Transfer

Occasionally, a firewall or NAT-based network may prevent the voice chat service from establishing its own network connection. Your application can implement an optional method in the client to provide real-time transfer of data between the participants. When your client implements the `voiceChatService:sendRealTimeData:toParticipantID:` method, if the voice chat service is unable to create its own real-time connection, it falls back and calls your method to transfer its data.

Starting a Chat

To start a voice chat, one of the participants calls the voice chat service's `startVoiceChatWithParticipantID:error:` method with the `participantID` of another participant. The service uses the client's network as described above to request a new chat.

When a service receives a connection request, the client's `voiceChatService:didReceiveInvitationFromParticipantID:callID:` method is called to handle it. The client accepts the chat request by calling the service's `acceptCallID:error:` method, or rejects it by calling `denyCallID:`. Your application may wish to prompt users to see if they want to accept the connection.

Once a connection has been established and accepted, the client receives a call to its `voiceChatService:didStartWithParticipantID:` method.

Disconnecting from Another Participant

Your application calls the service's `stopVoiceChatWithParticipantID:` method to end a voice chat. Your application should also stop the chat if it discovers that the other user is no longer available.

Controlling the Chat

Once the participants are connected, speech is automatically transmitted between the two iPhones. Your application can mute the local microphone by setting the service's `microphoneMuted` property, and it can adjust the volume of the remote participant by setting the service's `remoteParticipantVolume` property.

Your application can also enable monitoring of the volume level at either end of the connection. For example, you might use this to set an indicator in your user interface when a participant is talking. For local users, your application sets `inputMeteringEnabled` to `YES` to enable the meter, and reads the `inputMeterLevel` property to retrieve microphone data. Similarly, your application can monitor the other participant by setting `outputMeteringEnabled` to `YES` and reading the `outputMeterLevel` property. To improve application performance, your application should only enable metering when it expects to read the meter levels of the two participants.

Adding Voice Chat

Voice chat is implemented on top of a network connection provided by your application. The following example uses a `GKSession` object to provide a network to the client. For more information on `GKSession` objects, see "Peer to Peer Connectivity" (page 9). To implement voice chat, perform the following steps:

1. Configure the audio session to allow playback and recording.

    ```
    AVAudioSession *audioSession = [AVAudioSession sharedInstance];
    [audioSession setCategory:AVAudioSessionCategoryPlayAndRecord error:myErr];
    [audioSession setActive: YES error: myErr];
    ```

2. Implement the client's `participantID` method.

    ```
    - (NSString *)participantID
    {
        return session.peerID;
    }
    ```

    The participant identifier is a string that uniquely identifies the client. As a session's `peerID` string already uniquely identifies the peer, the client reuses it as the participant identifier.

3. Implement the client's `voiceChatService:sendData:toParticipantID:` method.

    ```
    - (void)voiceChatService:(GKVoiceChatService *)voiceChatService sendData:(NSData
     *)data toParticipantID:(NSString *)participantID
    {
        [session sendData: data toPeers:[NSArray arrayWithObject: participantID]
    withDataMode: GKSendDataReliable error: nil];
    }
    ```

    The service calls the client when it needs to send data to other participants in the chat. Most commonly, it does this to establish its own real-time connection with other participants. As both the `GKSession` and `GKVoiceChatService` use an `NSData` object to hold their data, simply pass it on to the session.

    If the same network is being used to transmit your own information, you may need to add an identifier before the packet to differentiate your data from voice chat data.

4. Implement the session's receive handler to forward data to the voice chat service.

    ```
    - (void) receiveData:(NSData *)data fromPeer:(NSString *)peer inSession:
    (GKSession *)session context:(void *)context;
    {
        [[GKVoiceChatService defaultVoiceChatService] receivedData:data
    fromParticipantID:peer];
    }
    ```

    This function mirrors the client's `voiceChatService:sendData:toParticipantID:` method, forwarding the data received from the session to the voice chat service.

5. Attach the client to the voice chat service.

```
MyChatClient *myClient = [[MyChatClient alloc] initWithSession: session];
[GKVoiceChatService defaultVoiceChatService].client = myClient;
```

6. Connect to the other participant.

```
[[GKVoiceChatService defaultVoiceChatService] startVoiceChatWithParticipantID:
 otherPeer error: nil];
```

Your application may want to do this automatically as part of the connection process, or offer the user an opportunity to create a voice chat separately. An appropriate place to automatically create a voice chat would be in the session delegate's `session:peer:didChangeState:` method.

7. Implement optional client methods.

If your application doesn't rely on the network connection to validate the other user, you may need to implement additional methods of the `GKVoiceChatClient` protocol. The `GKVoiceChatClient` protocol offers many methods that allow your client to be notified as other participants attempt to connect or otherwise change state.

Document Revision History

This table describes the changes to *Game Kit Programming Guide*.

Date	Notes
2009-05-28	Revised to include more conceptual material.
2009-03-12	New document that describes how to use GameKit to implement local networking over Bluetooth as well as voice chat services over any network.