Game Kit Programming Guide

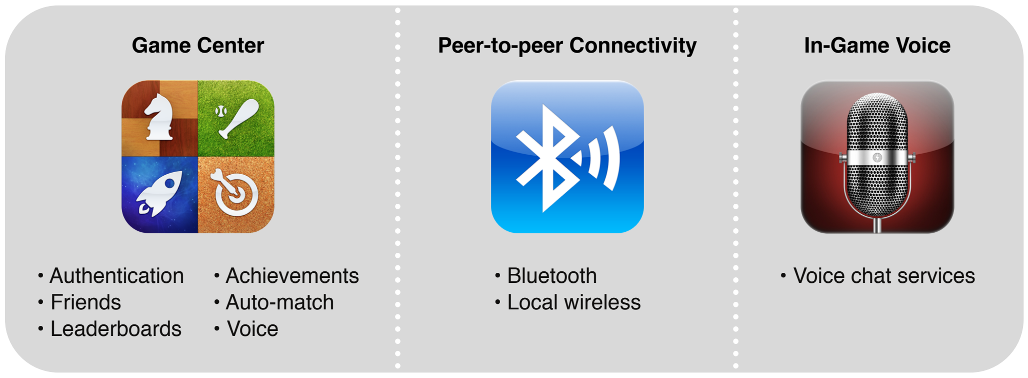
Developer

Contents

Figures, Tables, and Listings

About Game Kit

The Game Kit framework provides Objective C classes that help you to create great social games. Social games allow players to share their experiences with other players. When players tell their friends about their favorite games, this sharing encourages more players to download and play those games. Positive word-of-mouth recommendations from happy customers provide the best kind of free advertising for your game.



At a Glance

The Game Kit framework provides three distinct technologies: Game Center, Peer-to-Peer Connectivity, and In-Game Voice. You can adopt each technology in your application independently of the other technologies.

Game Center Provides a Centralized Network Service

Game Center is a social gaming service that allows players to share information about the games they are playing and to join other players in multiplayer matches. Game Center provides these services over both wireless and cellular networks. These key features are found Game Center:

- **Authentication** allows players to create a secured account on Game Center, and access it on iOS-based devices.
- **Friends** allow players to mark other players on Game Center as friends; friends have access to additional information about a player, including the games that player has played recently.
- **Leaderboards** allow your game to record and fetch player scores from Game Center.
- **Achievements** track a player's accomplishments in your game. Achievements are tracked by the Game Center service and can be viewed in the Game Center app as well as inside your game.

- **Auto-match** allows your game to create network games that connect players through Game Center. Players can invite their friends or be connected into a match with anonymous players. Players can receive invitations to play in a match even when your game is not running; your game is automatically launched to process the invitation.
- **Voice** allows your matches to provide voice communication between players connected to the match.

Game Center is supported in iOS 4.1 and later.

Relevant Chapters (Part I) "Game Center Overview" (page 13), "Working with Players in Game Center" (page 20), "Leaderboards" (page 29), "Achievements" (page 38), "Multiplayer" (page 48), "Using Matches to Implement Your Network Game" (page 63), and "Adding Voice Chat to a Match" (page 69)

Peer-to-Peer Connectivity Provides Local Wireless and Bluetooth Networking

Peer-to-peer connectivity allows your game to create an ad-hoc Bluetooth or local wireless network between multiple iOS-based devices. 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.

Peer-to-peer connectivity is provided in iOS 3.0 and later.

Relevant Chapters (Part II) "Peer-to-Peer Connectivity" (page 75), "Finding Peers with Peer Picker" (page 82) and "Working with Sessions" (page 85)

In-game Voice Provides a Common Voice Chat Infrastructure

In-game voice allows your application to provide voice communication between two iOS-based devices. In-game voice samples the microphone and handles mixing of audio playback for you. To use in-game voice, your application first establishes a network connection between the two devices. In-game voice uses this network connection to send its own handshaking information to create its own network connection.

In-game voice is provided in iOS 3.0 and later.

Relevant Chapters (Part III) "In-Game Voice" (page 88) and "Adding Voice Chat" (page 92)

How to Use This Document

If you want to add Game Center support to your game, first read "Game Center Overview" (page 13). Then read "Working with Players in Game Center" (page 20) to learn how to authenticate a player on the device. Once your game authenticates players, read the other chapters in Part I for the details on adding leaderboards, achievements, or matchmaking to your game.

If you want to add local area peer-to-peer networking to your application, read "Peer-to-Peer Connectivity" (page 75) for a conceptual overview of the `GKSession` and `GKPeerPickerController` classes. Then, read the remaining chapters in Part II for a walkthrough on how to add support for sessions to your application.

If your application already has a network connection and you want to add voice chat, read "In-Game Voice" (page 88) for an overview of the `GKVoiceChatService` class, and "Adding Voice Chat" (page 92) for a walkthrough on how to add support for the voice chat service to your application.

Prerequisites

Regardless of which Game Kit components you plan to use in your application, you should be familiar with Cocoa programming, especially delegation and memory management. Read *Cocoa Fundamentals Guide* for an introduction to Cocoa.

If you are designing a game that that works with Game Center, you should be comfortable with views and view controllers. Game Kit provides view controllers to present standard user interfaces for leaderboards, achievements, and matchmaking. See *View Programming Guide for iOS*. Game Center classes rely heavily on block objects to return results to your application. You should understand blocks and common block-programming techniques. See *A Short Practical Guide to Blocks* for an introduction to block programming.

Before implementing a network game using Game Kit, you should understand common network programming design patterns. Game Kit provides a low-level networking infrastructure, but your application must handle slow networks, disconnects, and security problems resulting from sending data over an unsecured network.

See Also

See *Game Kit Framework Reference* for details on specific Game Kit classes.

See iTunes Connect Developer Guide for details on how to configure your game in iTunes Connect.

The *GKTapper* sample demonstrates how to implement user authentication, leaderboards and achievements in a Game Center application.

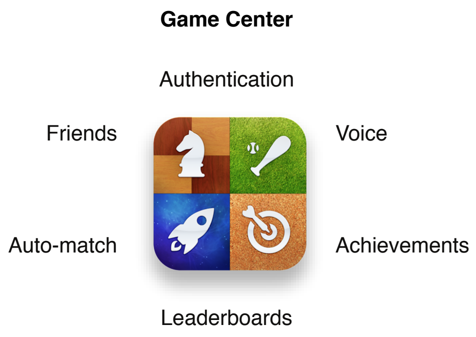
The *GKTank* sample demonstrates how to use peer-to-peer connectivity to implement a networked game.

Game Center

This part of *Game Kit Programming Guide* describes how your game work with Game Center.

Game Center Overview

This chapter describes Game Center's features and explains how to get started developing games that support Game Center.



What Is Game Center?

Game Center is a new social gaming network that is available on supported iOS-based devices running iOS 4.1 and later. A social gaming network allows players to share information about the games they are playing as well as to join other players in multiplayer games. Game Center provides these services over both wireless and cellular networks.

The functionality of Game Center is delivered by three interconnected components:

- The Game Center app is built into iOS 4.1 and later that provides a single place where players can access all of Game Center's features.
- The Game Center service is the online service that both your game and the Game Center app connect to. The online service stores data about each player and provides network bridging between devices on different networks.
- The Game Kit framework provides classes that you use to add support for Game Center to your game.

So what features does Game Center provide?

- **Authentication** allows each player to create and manage an online persona, also known as an **alias**. A player uses an alias to authenticate his or her identity on Game Center, to manage a list of friends and to post status messages that are visible to friends.

- **Leaderboards** allow your game to post scores to the Game Center service where they can later be viewed by players. Scores are viewed by launching the Game Center app; your game can also display a leaderboard with just a few lines of code. Leaderboards help foster competition between Game Center players.

  When you add leaderboards to your game, you decide how the game's scores are interpreted and displayed. For example, you can customize a score so that it appears as time, money, or an arbitrary value ("points"). You can choose to create multiple categories of leaderboards — for example, you might display a different leaderboard for each level of difficulty your game supports. Finally, your game can retrieve the scores stored in the leaderboard.

- **Achievements** are another way your game can measure a user's skills. An achievement is a specific goal that the user can accomplish within your game (for example, "find 10 gold coins" or "capture the flag in less than 30 seconds"). Your game customizes the text that describes an achievement, the number of points a player earns by completing an achievement, and the icon that is displayed for an achievement.

  As with leaderboards, a player can see their earned achievements within the Game Center app; they can also compare their achievements with those earned by a friend. Your game can display achievements with just a few lines of code, or you can download the achievement descriptions and use them to create a customized achievement screen.

- **Multiplayer** allows players interested in playing an online multiplayer game to discover each other and be connected into a match. Depending on your needs, your game can either use Game Center to connect all the participants together or have Game Center deliver a list of players to your game. In the latter case, you would provide your own network implementation inside your game that connects each device to a server you provide.

  Matchmaking provides both automated matching and invitations. When a user chooses automated matching, your game creates a request that specifies how many players should be invited into the match, and what types of players are eligible to join the match. Game Center then finds players that meet these criteria and adds them to the match. Invitations allow a player to invite their friends into a match, either from within your game or from within the Game Center app. When a friend is invited, the friend receives a push notification that allows him or her to launch your game and join the match. A push notification can be sent to a player even if the player does not have your game installed on the device. When this happens, the player can launch the App Store directly from the notification.

- **Peer-to-peer networking** is provided by the matchmaking service and provides a simple interface to send data and voice information to other participants in a match. Your game can create multiple voice channels, each with a different subset of the players in the match. For example, a game with multiple teams could create separate channels for each team allowing them to converse without allowing non-team members to listen in.

Supporting Game Center increases the visibility of your game. Players can see the games that their friends are playing, or invite friends into a match, giving those friends an opportunity to immediately purchase your game. By connecting game players to each other, you allow news to quickly spread about your game.

Essential Game Center Concepts

This section describes things you should know before adding Game Center support to your game.

All Game Center Games Start by Authenticating a Player

Matchmaking, leaderboards, and achievements all use an authenticated player on the device, known as the **local player**. For example, if your game reports scores to a leaderboard, they are always reported for the authenticated player. Most Game Center classes function only if there is an authenticated player. Before your game uses any Game Center features, it must authenticate the local player. When your game authenticates the local player, it checks to see if a player is already authenticated on the device. If there is not an authenticated local player, Game Kit displays a user interface to allow the player to log in with an existing account or create a new account.

Your game must disable all Game Center features when there is not an authenticated player.

Important On devices where Game Center is not available, Game Kit returns a `GKErrorNotSupported` error when your game attempts to authenticate the local player. Your game must handle any authentication errors by disabling all Game Center features.

Your Game Must Already Use a View Controller to Manage Its Interface

All Game Center views are presented modally by your game using your game's view controller as the hosting view controller. Leaderboards, achievements and matchmaking all provide view controller classes that allow your game to display relevant information to the user. For example, the `GKLeaderboardViewController` class provides a standard user interface to display your game's leaderboard information to the user.

Using a view controller to control your interface is a good idea for other reasons. For example, you use a view controller to support orientation changes on the device. For more information on using view controllers in your game, see *View Controller Programming Guide for iOS* .

Most Classes that Access Game Center Operate Asynchronously

With few exceptions, classes in Game Kit access the Game Center services asynchronously. These operations include posting data to Game Center, requesting data from Game Center, or asking Game Center to perform a task (such as matchmaking). In all cases, your game posts a request by calling Game Kit. Later, when the operation completes (or fails because of an error), Game Kit calls your game to return the results.

Callbacks for Game Center classes are implemented using block objects. The block object receives an error parameter. Depending on the operation your game requests, the block also receives other parameters to allow Game Kit to return information to your game. You should be comfortable with block objects before attempting to add Game Center to your game. See *A Short Practical Guide to Blocks* for an introduction to block programming.

Make Sure Your Game Sends Data Again After a Network Failure

When you implement leaderboards or achievements, your game reports a player's scores or achievement progress to Game Center. However, networking is never perfectly reliable. If a networking error occurs, your game should save the Game Kit object and retry the action again at a later time. Both the `GKScore` and `GKAchievement` classes support the `NSCoding` protocol, so your game can archive those objects when it moves into the background.

Your Game May Received Partial Data After a Network Failure

When your game retrieves data from Game Center (for example, when loading score data from a leaderboard), networking errors can occur. When a networking error occurs, Game Kit returns a connection error. However, where possible, Game Kit caches data it receives from Game Center. In this case, your game may receive both an error and partial results from Game Kit.

Steps for Implementing Game Center Awareness in Your Game

When you are ready to add Game Center support to your game, follow these steps:

1. Enable Game Center in iTunes Connect. See "Configure Your Application in iTunes Connect" (page 17).
2. Configure the Bundle Identifier for your game. See "Create Your Application's Bundle Identifier" (page 17).
3. Link to the Game Kit framework.
4. Import the `GameKit/GameKit.h` header.
5. Determine whether your game requires Game Center.
   - If your game requires Game Center, add the Game Center key to the list of capabilities your application requires from the device. See "Requiring Game Center in Your Game" (page 17).
   - If your game does not require Game Center, you should establish a weak link from your game to the Game Kit framework. Then, when your game launches, test to make sure that Game Center is supported. See "Optionally Using Game Center In Your Game" (page 17).
6. Authenticate the player as soon as your application has launched far enough to present a user interface to the user. See "Working with Players in Game Center" (page 20).

Configure Your Application in iTunes Connect

Before you add any Game Center code to your application, you need to set up your game in iTunes Connect and enable it to use Game Center. This is also where you go to configure your application's leaderboards and achievement information.

The process for configuring your game to support Game Center is described in iTunes Connect Developer Guide.

Create Your Application's Bundle Identifier

A **bundle identifier** is a string you create for an application (such as `com.myCompany.myCoolGame`). The bundle identifier is used throughout the application development process to uniquely identify an application. For example, the bundle identifier is used to match the application you built in Xcode to the data you configured in iTunes Connect.

For applications that do not use Game Center, you need only provide a bundle identifier in your Xcode project prior to shipping your application to customers; you can start developing an application without setting a bundle identifier. However, when you develop a Game Center application, you must set the bundle identifier before implementing Game Center in your application. Game Center uses the application's bundle identifier to retrieve the application's data from the Game Center service. Your application cannot use matchmaking or retrieve leaderboard or achievement information until you set the application's bundle identifier in your project.

For more information on setting the bundle identifier, see *Information Property List Key Reference*.

Requiring Game Center in Your Game

If your game requires Game Center to function (for example, a multiplayer game that requires Game Center to match players), you want to ensure that only devices that support Game Center can download your application. To ensure that your application only runs on supported devices, add the `gamekit` key to the list of required device capabilities in your application's `Info.plist` file. See "iTunes Requirements" in *iOS App Programming Guide*.

Optionally Supporting Game Center In Your Game

If you want your game to use Game Center, but your game does not require Game Center in order to function properly, you can weak link your game to the Game Kit framework and test for its existence at runtime.

Listing 1-1 (page 18) provides code you should use to test for Game Center support. This function tests for the existence of the `GKLocalPlayer` class, which is required to perform player authentication, and also for iOS 4.1, which is the first version of iOS that has complete support for Game Center.

Note Game Center was first previewed in iOS 4.0, and some Game Center classes can be found there. However, the classes changed prior to their official release in iOS 4.1. You should not attempt to use Game Center on devices running iOS 4.0.

Listing 1-1 Testing whether the Game Center classes are available

```
BOOL isGameCenterAPIAvailable()
{
    // Check for presence of GKLocalPlayer class.
    BOOL localPlayerClassAvailable = (NSClassFromString(@"GKLocalPlayer")) != nil;

    // The device must be running iOS 4.1 or later.
    NSString *reqSysVer = @"4.1";
    NSString *currSysVer = [[UIDevice currentDevice] systemVersion];
    BOOL osVersionSupported = ([currSysVer compare:reqSysVer options:NSNumericSearch]
 != NSOrderedAscending);

    return (localPlayerClassAvailable && osVersionSupported);
}
```

iOS 4.1 may be installed on some devices that do not support Game Center. On those devices, the `isGameCenterAPIAvailable` function defined Listing 1-1 (page 18) still returns `YES`. The next step in confirming whether Game Center may be used on the device is to attempt to authenticate the local player; on devices that do not support game center, your game receives a `GKErrorNotSupported` error. For more information, see "Authenticating a Local Player" (page 23).

Testing a Game Center Application

To help you test your application, Apple provides a sandbox environment for Game Center. This sandbox environment duplicates the live functionality of Game Center, but is separate from the live servers. The sandbox allows you to test your Game Center features without making your application visible to regular users. You should thoroughly test your Game Center application in sandbox before submitting your application for approval.

As a developer, you are required to create a separate Game Center account for Sandbox. At any given time, you must choose whether to log into Sandbox for testing, or into the live environment. Start by launching the Game Center app and logging out the currently authenticated player. After this, run your game or another

Game Center enabled application. Depending on how that application is distributed, you enter different credentials. If that app is provisioned for development, enter your test account information (logging you into the Sandbox). Otherwise, enter your live account information (logging you into the live environment. Table 1-1 shows which builds run in which environments.

Important Always create new test accounts specifically to test your application in Game Center. Never use an existing Apple ID.

Table 1-1 Which build for which audience and environment

Application	Audience	Game Center Environment
Simulator build	Developer	Sandbox Environment
Developer build	Developer	Sandbox Environment
Ad-hoc distribution build	Beta Testers	Sandbox Environment
Signed Distribution build	End Users	Live Environment

The Sandbox does not allow sharing of information about what games are being played. This prevents your testers from revealing the existence of your application to other players.

Testing Your Game in Simulator

Leaderboards and achievements work the same way in Simulator as they do on a device. However, matchmaking invitations may not be sent or received while your application is running in Simulator.

Working with Players in Game Center

Players are the bedrock of Game Center. Players join online games, rack up high scores, and achieve great results in your games. But before you can arrange for players to do those things in your game, you need to understand how Game Center works with players.

A player that wants to take advantage of Game Center must first create a Game Center account. A Game Center account serves many roles: it allows the player to be authenticated by Game Center, it uniquely identifies the player, and it allows data specific to that player to be stored on Game Center and accessed by your application. Such data includes:

- Every score in a leaderboard earned by that player
- The player's progress on achievements
- A player's list of friends

In this chapter, you'll learn how to manage player information in your application. In particular, you'll learn:

- How your game identifies different players on Game Center
- The proper steps to log a player into Game Center
- How to retrieve details about players from Game Center
- How to allow a player to invite other players to become friends on Game Center

Player Identifier Strings are Used by Your Game To Identify Players

Game Center assigns to each Game Center account a unique string, known as a **player identifier**, that identifies that specific account. Although a player can change other information associated with his or her account, an account's player identifier never changes. Player identifiers are used throughout the Game Kit framework whenever specific players need to be identified. For example, if your application uses Game Center to create networked matches, each device in the match is identified by the player identifier for the player logged into that device. When your application running on one device sends data to other participants, it addresses the network messages using the player identifiers.

In addition to using player identifiers in your interactions with Game Center, your application should also use player identifiers whenever you need to associate information with a specific player. For example, if your game stores data on a player's device to track the player's progress in your game, you should use player identifiers to distinguish between multiple players playing on the same device. Similarly, if your application stores information about players on your own network server, your server can use player identifiers to uniquely identify each player's data.

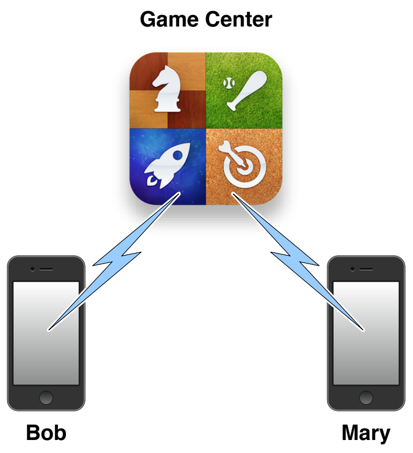
Important Never make assumptions about the format or length of player identifier strings. Although any individual player identifier string is immutable, the format and length of new player identifier strings are subject to change.

Player identifiers are intended only to be used internally by your game or Game Center. Player identifiers should never be displayed to the player. However, displaying a name for a player is important. Every player gets to choose their own name, also known as their **alias**. Instead of displaying a player identifier, always retrieve a player's alias from Game Center instead.

The Local Player Is the Player Signed into the Device

For an instance of your application running on an iOS-based device, one player takes precedence over other players. The **local player** is the player that is currently authenticated to play on a particular device. For example, in Figure 2-1, two players are connected in a match. On the left device, Bob is the local player and Mary is a remote player. On the right device, Mary is the local player and Bob is a remote player.

Figure 2-1 Local and remote players



For any given device, only one player can be connected to Game Center at a time. A player connects to Game Center either by launching the Game Center app or by launching a game that implements Game Center support. In either case, they are presented with an interface to provide an account name and password. After authenticating their account on a device, that player remains connected on that device, even when the player switches to other applications or reboots the device. A player only disconnects from Game Center only by launching the Game Center application and explicitly signing out.

Important Games that support multitasking should take special note of this behavior. When your game moves into the background, the player may launch the Game Center application and sign out. Also, another player might sign in before control is returned to your application. Whenever your game moves to the foreground, it may need to disable its Game Center features when there is no longer an authenticated player or it may need to refresh its internal state based on the identity of the new local player.

Most classes in Game Kit that access Game Center expect the device to have an authenticated local player and work on that player's behalf. For example, when your game reports scores to a leaderboard, it can only report scores earned by the local player. If your application attempts to report scores to a leaderboard (or other similar tasks) when there is not an authenticated local player, Game Kit returns an error to your game.

A Player Object Provide Details About a Player

If your game knows the player identifier for an account on Game Center, it can retrieve some information about that player using Game Kit. Game Kit provides a player's details to your application by returning a `GKPlayer` object. The player object includes the following properties:

- The `playerID` property holds the player's identifier string.
- The `alias` property holds the user-visible string chosen by this player.
- The `isFriend` property states whether the player is a friend of the current local player.

For more information about retrieving details about a player, see "Retrieving Details for Game Center Players" (page 26).

The `GKLocalPlayer` class is a special subclass of the `GKPlayer` class, and includes additional properties for the local player as well as methods to authenticate the local player. The `GKLocalPlayer` class provides additional properties:

- The `friends` property is populated with the identifier strings for other players on Game Center that are marked as the local player's friends. This property is populated only after your application specifically requests those identifiers from Game Center. See "Retrieving Identifiers for a Local Player's Friends" (page 27).
- The `underage` property states whether this player is underage. Some Game Center features are disabled when the value of this property is `YES`. This property is provided so that your application has the option to disable its own features for an underage player.

Authenticating a Local Player

All games that support Game Center must authenticate the local player before using any of Game Center's features. Your game should authenticate the player as early as possible after launching. Ideally, authentication should happen as soon as your game can present a user interface to the player. When your game authenticates a player, Game Kit first checks to see whether there is already an authenticated player on the device. If there is an authenticated player, Game Kit briefly displays a welcome banner to the player. If there is not presently an authenticated player on the device, Game Kit displays an authentication dialog that allows the player to either sign in with an existing account or to create a new Game Center account. So, when your game authenticates the local player, it also transparently allows players to create Game Center accounts.

Game Kit also transparently handles when the player decided to opt out of using Game Center. After a player dismisses the Game Center authentication dialog a few times, Game Kit silently returns control to your application without displaying the authentication dialog. After Game Kit disables the authentication dialog, the player creates an account only within the Game Center app.

Important Game Kit handles opting out of Game Center across all games that support Game Center. This means that if a player has already declined to create an account on other games, they may never see the dialog in your game. Because Game Kit handles this process across all games, your game should attempt to authenticate the player on launch every time it launches. Do not implement your own mechanism to disable Game Center authentication in your game.

Listing 2-1 shows an implementation of a method that authenticates the local player. Typically, such a method would be implemented in your application delegate, and called from inside the application delegate's `application:didFinishLaunchingWithOptions:` method. The method retrieves the shared instance of the `GKLocalPlayer` class and calls that object's `authenticateWithCompletionHandler:` method. Authenticating the local player happens asynchronously; your application continues to launch after initiating the player authentication. When authentication completes, Game Kit calls the block object provided by your game.

Listing 2-1 Authenticating the local player

```
- (void) authenticateLocalPlayer
{
    GKLocalPlayer *localPlayer = [GKLocalPlayer localPlayer];
    [localPlayer authenticateWithCompletionHandler:^(NSError *error) {
         if (localPlayer.isAuthenticated)
         {
             // Perform additional tasks for the authenticated player.
         }
    }];
}
```

Your application should always check the `isAuthenticated` property on the local player object to determine whether Game Kit was able to authenticate a local player. Do not rely on the error received by your application to determine whether an authenticated player is available on the device. Even when an error is returned to your application, Game Kit may have sufficient cached information to provide an authenticated player to your game. Also, it is not necessary for your application to display errors to the player when authentication fails; Game Kit already displays the most common errors to the player on your behalf. Most authentication errors are useful primarily to assist in debugging your application.

Two common errors that may be received by your game are worth describing in more detail:

- If your application receives a `GKErrorGameUnrecognized` error, you have not enabled Game Center for your application in iTunes Connect. Log into your iTunes Connect account and verify that your application has Game Center enabled. Also, confirm that the bundle identifier in your Xcode project matches the bundle identifier you assigned to your application in iTunes Connect.
- If your application receives a `GKErrorNotSupported` error, the device your application is running on does not support Game Center. You should disable further attempts to authenticate a player on this device.

If `isAuthenticated` is `YES`, then authentication has been completed successfully. Your game can now read other properties on the local player object and use other classes that access Game Center. For example, here are common tasks that most games should perform after successfully authenticating the player:

- Read the local player object's `alias` property to retrieve the local player's nickname. Use this nickname throughout your game when you want to refer to the player; do not prompt the player separately for their name.
- Add an invitation handler to receive matchmaking requests. If your game supports matchmaking, it should always add an invitation handler immediately authenticating the local player. Your game may have been launched by iOS specifically to receive a pending invitation; adding the invitation handler immediately after authenticating the player allows that invitation to be processed promptly by your game. See "Processing Invitations from Other Players" (page 52)
- Retrieve the local player's previous progress on achievements. See Listing 4-2 (page 41).
- Retrieve a list of player identifiers for the local player's friends. This is the first step in loading more detailed information about those players. See "Retrieving Identifiers for a Local Player's Friends" (page 27).
- If your game stores its own custom information for a particular player (such as state variables indicating the player's progress through your game), your completion handler might also load this data so that it can restore the player's progress.

Authenticating the Local Player in a Multitasking Application

In iOS 4 and later, multitasking allows a user to move the frontmost application into the background in order to allow a different application to move to the foreground. Multitasking allows users to quickly switch between applications, as well as allowing some applications to continue processing in the background. A game that supports multitasking and Game Center must take additional steps when authenticating the local player. When your game is in the background, the status of the authenticated player may change. A player can use the Game Center app to sign out. Another player might sign in before your game returns to the foreground.

If your game supports multitasking, Game Kit calls your completion handler as described in "Authenticating a Local Player," but it also retains your completion handler for later use. Each time your application is moved from the background to the foreground, Game Kit automatically authenticates the local player again on your behalf and calls your completion handler to provide updated information about the state of the authenticated player.

Important Automatically authenticating the local player after moving to the foreground is provided only on iOS 4.2 and later. On iOS 4.1, there is no correct way to authenticate the local player after your application moves to the background.

Here are some guidelines for authenticating the local player in a game that supports multitasking:

- Like other multitasking applications, your game should archive its state before moving into the background. This state includes objects created using classes provided by Game Kit, such as score or achievement objects, that are associated with the current local player. Your game archives its state because it may be jettisoned from memory after moving into the background.
- As soon as your game moves to the background, the value of the local player object's `isAuthenticated` property becomes invalid and remains invalid until your game moves back to the foreground, Game Kit authenticates the player, and your authentication handler is called. Your game must act as though there is not an authenticated player until your completion handler is called. Once your handler is called, the `isAuthenticated` property is valid again, and holds the whether a player is authenticated on the device.
- If the value of the `isAuthenticated` property changed to `NO`, then there is no longer a local player authorized to access Game Center content. Your game should dispose of any Game Kit objects for the previous local player it still has in memory. However, your game should be prepared to load the archived state on a future launch or transition to the foreground. If a player logs into the device at some future time, your game can unarchive objects associated with that player and use them to allow the player to resume play.
- If the value of the `isAuthenticated` property is `YES`, then there is a local player authenticated on the device. However, when your game was in the background, the player logged into Game Center may have changed. Your application should store the player identifier for the local player in an instance variable after it authenticates the player. On future calls to your completion handler, compare the value of the local

player object's `playerID` property to the identifier previously stored by your application. If the identifier has changed, the player has changed. Your game should dispose of any objects associated with the previous player and then create or load the appropriate state for the new local player.

Retrieving Details for Game Center Players

Many Game Kit classes return player identifiers to your application. For example, every score provided by a leaderboard uses a player identifier to identify the player who earned the score. If your game has a list of player identifiers, it can load details for those players by calling the `loadPlayersForIdentifiers:withCompletionHandler:` class method on the `GKPlayer` class. Listing 2-2 shows a typical implementation. This Game Kit method takes two parameters. The first is an array of player identifiers; the second is a completion handler to be called after Game Kit retrieves the data from Game Kit. Game Kit loads the data asynchronously in the background and calls your completion handler after the task completes. The completion handler receives an array of `GKPlayer` objects (one for each identifier) as well as an error parameter.

Listing 2-2 Retrieving a collection of player objects

```
- (void) loadPlayerData: (NSArray *) identifiers
{
    [GKPlayer loadPlayersForIdentifiers:identifiers withCompletionHandler:^(NSArray
 *players, NSError *error) {
        if (error != nil)
        {
            // Handle the error.
        }
        if (players != nil)
        {
            // Process the array of GKPlayer objects.
        }
     }];
}
```

If Game Kit was unable to load information for all of the players, it provides an error to the completion handler. When this occurs, the `players` parameter may provide a partial array for the players that Game Kit was able to obtain information about. For this reason, Listing 2-2 tests the error condition separately from processing the array of player objects.

Working with Friends

Game Center is intended to be a social experience. Game Center allows a player to invite other players to be friends. When two players are friends, they can see each others status in the Game Center app, compare scores, and invite each other into matches more easily. Through Game Kit, your application can also access some information about a local player's friends or allow the player to invite players to become friends. For example, your application might use this functionality to allow a player to send a friend invitation to a player they just met in a match played within your game.

Retrieving the Local Player's Friends

Retrieving details about the local player's friends is a two-step process. First, your game loads the `friends` property of the local player object. Then, as with any other player identifier, your game loads the details for those players from Game Center.

Listing 2-3 shows how your game loads the list of player identifiers for the local player's friends. It then calls the method defined in Listing 2-2 (page 26) to fetch the details for those players.

Listing 2-3 Retrieving a local player's friends

```
- (void) retrieveFriends
{
   GKLocalPlayer *lp = [GKLocalPlayer localPlayer];
   if (lp.authenticated)
   {
      [lp loadFriendsWithCompletionHandler:^(NSArray *friends, NSError *error) {
         if (friends != nil)
         {
            [self loadPlayerData: friends];
         }
      }];
   }
}
```

Sending Friend Invitations to Other Players

The Game Center app allows players to send invitations to other players. In iOS 4.2 and later, your game may use the `GKFriendRequestComposeViewController` class to allow a player to send friend requests. The friend request must be presented modally by a view controller that your game creates.

Listing 2-4 shows how your view controller might allow a player to send a request to other players. For this method, an array of player identifiers is passed in as a parameter. The method instantiates a `GKFriendRequestComposeViewController` object, sets its delegate, and adds the list of players intended to receive the invitation. The view controller then presents the friend request and returns.

Listing 2-4 Displaying a friend request

```
- (void) inviteFriends: (NSArray*) identifiers
{
    GKFriendRequestComposeViewController *friendRequestViewController =
[[GKFriendRequestComposeViewController alloc] init];
    friendRequestViewController.composeViewDelegate = self;
    if (identifiers)
    {
        [friendRequestViewController addRecipientsWithPlayerIDs: identifiers];
    }
    [self presentModalViewController: friendRequestViewController animated: YES];
    [friendRequestViewController release];
}
```

When the player dismisses the friend request, the delegate's `friendRequestComposeViewControllerDidFinish:` method is called. Listing 2-5 (page 28) shows how your view controller should dismiss the request.

Listing 2-5 Responding when the player dismisses the friends request

```
-
(void)friendRequestComposeViewControllerDidFinish:(GKFriendRequestComposeViewController
 *)viewController
{
    [self dismissModalViewControllerAnimated:YES];
}
```

Leaderboards

Many games offer scoring systems to allow players to measure how well they have mastered the game's rules. As a player's skill improves, his or her scores also improve. In Game Center, a leaderboard is used to record scores earned by anyone who plays your game. Your game posts player scores to Game Center. When a player wants to see their scores, they bring up a leaderboard screen, either in the Game Center application or by viewing the leaderboard inside your game. Those scores can be filtered; for example, the leaderboard screen can be customized to show scores earned by a player's friends.

Checklist for Supporting Leaderboards

To add leaderboards to your game, you need to do the following:

- Before you can add leaderboards, you need to have added code to your game to authenticate local players. See "Working with Players in Game Center" (page 20).
- Decide how you want your game to calculate scores. You are free to design your own scoring mechanisms that are appropriate to your game.
- Go to iTunes Connect and configure one or more leaderboards for your application. In this step, you decide how you want the scores that your application sends to Game Center to be formatted when they are displayed in a leaderboard. Leaderboards are fully localized for different languages and regions. See "Configuring a Leaderboard on iTunes Connect" (page 30).
- Add code to report scores to Game Center. See "Posting Scores to Game Center" (page 32).
- Add code to handle reporting problems. If there is a condition that makes your application unable to report a score to Game Center, your application should archive the score object and attempt to resend it when the condition that caused the error ends. See "Recovering from Score-Reporting Errors" (page 33).
- Add code to allows a player to view a leaderboard inside your application. To display a leaderboard that looks similar to the leaderboard displayed by the Game Center application, see "Showing the Standard Leaderboard" (page 33). Optionally, you can retrieve the score data from Game Center and use it to customize your own user interface. Retrieving score data is described in "Retrieving Leaderboard Scores" (page 34).

Configuring a Leaderboard on iTunes Connect

To Game Center, a score is just a 64-bit integer value reported by your application. You are free to decide what a score means, and how your application calculates it. When you are ready to add leaderboards to your application, you configure leaderboards on iTunes Connect so that Game Center knows how scores should be formatted and displayed to the player. Further, you provide localized strings so that the scores can be displayed correctly in different languages. A key advantage of configuring leaderboards in iTunes Connect is that the Game Center application can show your game's scores without your having to write any code.

Each game has its own leaderboard configuration in iTunes connect; leaderboard configurations and score data are never shared between games.

Configuring a leaderboard requires you to make some decisions:

- What units do you want your scores measured in?
- Should score values be ranked in ascending or descending order?
- How should scores be formatted?
- Do you want scores reported by your application to be included in a single leaderboard, or do you want different modes of play in your game to have separate leaderboards?

The remainder of this section is an overview of the decisions you need to make when creating a leaderboard. For more information on configuring your leaderboards in iTunes Connect, see iTunes Connect Developer Guide.

Defining Your Score Format

Your application is free to calculate scores any way it wants, but you need to provide enough information so that they can be formatted and displayed to the user. In Game Center, scores are measured in one of three ways: as an abstract measurement, as a time value, or as a monetary value. You decide on a specific formatting type and then provide localized strings for the units. For example, a game might choose `Integer` as the formatting type and " point" and " points" as the english localization for singular and plural values of that score. If you later reported a score of 10, the score would be formatted as "10 points". Other games might use the same formatting type, but provide different localized strings (" laps", " cars").

Note A leading space is not added to the suffix by default. If you want a leading space after the value, you must add it when you define your suffixes.

When you define the score format, you choose whether scores are ranked in ascending or descending order. For example, a racing game that records scores as the time required to complete a track would rank scores in ascending order; the lowest score (fastest time) is the best score. A game that measures scores in the amount of wealth earned would choose descending order. The best score is always displayed at the top of the list.

Leaderboard Categories

Games often provide multiple configurations of game play. For example, your game might have multiple difficulty levels (easy, medium, hard), provide different rules for playing or scoring the game, or even different locations (maps, levels) within the game itself. So, when configuring your game on Game Center, consider whether your game should support a single leaderboard or separate leaderboards for each rules configuration.

Each leaderboard you configure in iTunes Connect is identified using a **category**, a string that uniquely identifies that leaderboard. You create your own category identifier for a leaderboard when you configure the leaderboards other settings, such as the localized title for the leaderboard, the score values, and how scores are formatted. When you implement leaderboard scoring in your game, your game uses the a category string to route the scores to the appropriate leaderboard. For example, if your game has three different levels of difficulty, you might create one leaderboard for each level of difficulty, using three category strings: “mygame.easy”, “mygame.medium” and “mygame.hard”.

Each game may create up to 25 categories. One category is always defined as the **default** category; if your application reports a game score without specifying a category, it is added to the list of scores associated with the default category.

Aggregate Leaderboards

An aggregate leaderboard is a special type of leaderboard that combines the scores recorded for a set of other leaderboards. When a player views the aggregate leaderboard, the aggregate leaderboard takes the scores recorded in any of the other leaderboards and sorts and ranks them as a single collection of scores. For example, if your game offered multiple maps to play on, you might create a separate leaderboard for each map as well as a leaderboard that shows scores earned on any map.

When deciding whether to set up one or more aggregate leaderboards for your game, keep the following guidelines in mind:

- The leaderboards combined to form an aggregate leaderboard must share the same scoring format and sort order.
- A leaderboard may only be part of a single aggregate leaderboard.

- Aggregate leaderboards may not be combined into another aggregate leaderboard..

Reporting Scores to Game Center

Your application transmits scores to Game Center by creating a `GKScore` object. A score object has properties for the player that earned the score, the date and time the score was earned, the category for the leaderboard the score should be reported to, and the score that was earned. Your application configures the score object and then calls its `reportScoreWithCompletionHandler:` method to send the data to Game Center.

Listing 3-1 shows how to use a score object to report a score to Game Center.

Listing 3-1 Reporting a score to Game Center

```
- (void) reportScore: (int64_t) score forCategory: (NSString*) category
{
    GKScore *scoreReporter = [[[GKScore alloc] initWithCategory:category]
autorelease];
    scoreReporter.value = score;

    [scoreReporter reportScoreWithCompletionHandler:^(NSError *error) {
       if (error != nil)
       {
            // handle the reporting error
        }
    }];
}
```

The score object is initialized with the category identifier for the leaderboard it should be reported to, then the method sets the value property to the score the player earned. The category identifier must be one of the category identifiers you defined when you configured your leaderboards in iTunes Connect. This code for reporting a score does not set the player who earned the score or the time the score was earned; those properties are set automatically when the score object was created. Scores are always reported for the local player.

Recovering from Score-Reporting Errors

If your application receives a network error, you should not discard the score. Instead, store the score object and attempt to report the player's process at a later time. `GKScore` objects support the `NSCoding` protocol, so if necessary, they can be archived when your application terminates and unarchived after it launches.

Showing the Standard Leaderboard

In addition to sending scores to Game Center, your should also allow players to view their scores from within your app. The simplest way to do this is with a `GKLeaderboardViewController` object. A leaderboard view controller provides a user interface similar to the leaderboard found in the Game Center application. The leaderboard view controller is presented modally by a view controller that you implement in your application.

Listing 3-2 provides a method your view controller can use to display the default leaderboard. The method instantiates a new leaderboard view controller and sets its `leaderboardDelegate` property to point to your view controller. The view controller then presents the leaderboard and waits for the delegate to be called.

Listing 3-2 Displaying the default leaderboard

```
- (void) showLeaderboard
{
    GKLeaderboardViewController *leaderboardController =
[[GKLeaderboardViewController alloc] init];
    if (leaderboardController != nil)
    {
        leaderboardController.leaderboardDelegate = self;
        [self presentModalViewController: leaderboardController animated: YES];
    }
}
```

You can configure the leaderboard view controller's `category` and `timeScope` properties before presenting the leaderboard:

- The `category` property allows you to configure which category screen is displayed when the leaderboard opens. If you do not set this property, the leaderboard opens to the default category you configured in iTunes Connect.

- The `timeScope` property allows you to configure which scores are displayed to the user. For example, a time scope of `GKLeaderboardTimeScopeAllTime` retrieves the best scores regardless of when they were scored. The default value is `GKLeaderboardTimeScopeToday`, which shows scores earned in the last 24 hours.

When the user dismisses the leaderboard, the delegate's `leaderboardViewControllerDidFinish:` method is called. Listing 3-3 (page 34) shows how your view controller should dismiss the leaderboard.

Listing 3-3 Responding when the player dismisses the leaderboard

```
- (void)leaderboardViewControllerDidFinish:(GKLeaderboardViewController
*)viewController
{
    [self dismissModalViewControllerAnimated:YES];
}
```

You may want to save the leaderboard view controller's `category` and `timeScope` properties before disposing of the leaderboard view controller. These properties hold the values of the last selections the player chose while viewing the leaderboards. You can then use those same values to initialize the leaderboard view controller the next time the player wants to see the leaderboard.

Retrieving Leaderboard Scores

If you want your application to examine the score data or create a custom leaderboard view, you can have your application directly load score data from Game Center. To retrieve score data, your application uses the `GKLeaderboard` class. A `GKLeaderboard` object represents a query for data stored on Game Center for your application. To load score data, your application creates a `GKLeaderboard` object and sets its properties to filter for a specific set of scores. Your application calls the leaderboard object's `loadScoresWithCompletionHandler:` method to load the scores. When the data is loaded from Game Center, Game Kit calls the block you provided.

Listing 3-4 shows a typical leaderboard data query. The method for this query initializes a new leaderboard object and configures the `playerScope`, `timeScope`, and `range` properties to grab the top ten scores earned by anyone playing your game, regardless of when the scores were reported.

Listing 3-4 Retrieving the top ten scores

```
- (void) retrieveTopTenScores
```

```
{
    GKLeaderboard *leaderboardRequest = [[GKLeaderboard alloc] init];
    if (leaderboardRequest != nil)
    {
        leaderboardRequest.playerScope = GKLeaderboardPlayerScopeGlobal;
        leaderboardRequest.timeScope = GKLeaderboardTimeScopeAllTime;
        leaderboardRequest.range = NSMakeRange(1,10);
        [leaderboardRequest loadScoresWithCompletionHandler: ^(NSArray *scores,
NSError *error) {
            if (error != nil)
            {
                // handle the error.
            }
            if (scores != nil)
            {
                // process the score information.
            }
            }];
    }
}
```

Your application can create a leaderboard request that explicitly provides the identifiers for the players whose scores you are interested in. When you provide a list of players, the `playerScope` property is ignored. Listing 3-5 shows how to use the player identifiers in a list associated with a match in order to load the players' best scores.

Listing 3-5 Retrieving the top scores for players in a match

```
- (void) receiveMatchBestScores: (GKMatch*) match
{
    GKLeaderboard *query = [[GKLeaderboard alloc] initWithPlayerIDs:
match.playerIDs];
    if (query != nil)
    {
        [query loadScoresWithCompletionHandler: ^(NSArray *scores, NSError *error)
 {
            if (error != nil)
            {
                // handle the error.
            }
            if (scores != nil)
            {
                // process the score information.
            }
        }];
    }
}
```

In either case, the returned `GKScore` objects provide the data your application needs to create a custom view. Your application can use the score object's `playerID` to load the player's alias, as described in "Retrieving Details for Game Center Players" (page 26). The `value` property holds the actual value you reported to Game Center. More importantly, the `formattedValue` property provides a string with the that actual value formatted according to the parameters you provided in iTunes Connect.

Important You may be tempted to write your own formatting code rather than using the `formattedValue` property. Do not do this. Using the built-in support makes it easy to localize the score value into other languages, and provides a string that is consistent with the presentation of your scores in the Game Center application.

To maintain an optimal user experience, your app should only query the leaderboard for data it needs to use or display. For example, do not attempt to retrieve all the scores stored in the leaderboard at once. Instead, grab smaller portions of the leaderboard and update your views as necessary.

Retrieving Category Titles

If your application presents a custom leaderboard screen, your application also needs the localized titles of the categories you configured in iTunes Connect. Listing 3-6 (page 36) shows how your application can load the titles from Game Center. As with most other classes that access Game Center, this code returns the results to your application by calling the block object you provided.

Listing 3-6 Retrieving category titles

```
- (void) loadCategoryTitles
```

```
{
    [GKLeaderboard loadCategoriesWithCompletionHandler:^(NSArray *categories,
NSArray *titles, NSError *error) {
        if (error != nil)
        {
            // handle the error
        }
        // use the category and title information
     }];
}
```

The data returned in the two arrays in Listing 3-6 (page 36) are the category identifiers and their corresponding titles.

Achievements

Achievements allow your application to create goals for players. By naming a goal and offering visual recognition when a player achieves that goal, you motivate the player and give them something to share with their friends. Game Center allows players to view their progress in the Game Center application, and you can use Game Kit to provide the same progress information inside your application.

Checklist for Supporting Achievements

To add achievements to your application, you need to take the following steps:

- If you haven't already done so, add code to your application to authenticate the local player. See "Working with Players in Game Center" (page 20).
- Go to iTunes Connect and configure the achievements for your game. You provide all of the data needed to display achievements to the player. See "Designing Achievements for Your Game" (page 38).
- Add code to your game to report the local player's progress to Game Center. By storing the progress on Game Center, you also make the player's progress visible in the Game Center application. See "Reporting Achievement Progress to Game Center" (page 40).
- Add code to load the local player's previous progress on achievements from Game Center. The local player's progress on achievements should be loaded shortly after you successfully authenticate them. See "Loading Achievement Progress" (page 41).
- Add code to allow the player to view achievements from within your application. See "Displaying Achievements Using an Achievement View Controller" (page 45) to see how your application can display the standard achievements screen. Or, if you want to customize how achievements are shown to match your application's user interface, read "Creating a Custom Achievement User Interface" (page 46) to learn how your game can load achievement descriptions from Game Center.

Designing Achievements for Your Game

When you add a new achievement, you should first decide how you want the player to accomplish that goal within your game. What those goals are and how those goals are accomplished vary for different games. Here are a couple of guidelines to follow when designing your achievements:

- Create achievements for different sections of your game. For example, if your game has different modes of play, make sure to add achievements for each play mode. This encourages the player to try each part of your game.
- Create achievements for a wide range of player skill levels. New players should be able to earn achievements while some achievements should only be earned my skilled players through significant effort. Having different levels of achievements encourages players to improve their skills. When they achieve difficult goals, your game acknowledges their achievements.

When you are ready to create an achievement, you need to define the achievement in iTunes Connect by creating a description.

An achievement description includes the following pieces of information:

- An **identifier** string that uniquely identifies the achievement. You choose this identifier string, and your application this string when reports progress on an achievement. Identifiers are never shown to the player.
- A **title** string that names the achievement.
- Two **description** strings for the achievement. Your descriptions should clearly explain the achievement to the player. The first description is used when the user has not completed the achievement and should clearly explain what the player must do to earn the reward. The second description is used after the user earns the achievement and should clearly state what they accomplished.
- An integer value for the number of **points** earned by completing this achievement. An achievement that is more difficult or more time consuming to earn should be worth more points. Each game has a budget of 1000 points to divide between its achievements. No achievement may reward more than 100 points. If your game supports additional content through the in-app purchase feature, you may want to reserve some of your point budget for achievements that apply to the In App Purchase in-app purchase content.
- An **image** to be displayed when a player completes the achievement. You create an image for the completed achievement only. Incomplete achievements always display a standard image provided by Game Center.
- Whether the achievement is visible to the user when they first launch your game or if it must be discovered during play. Most achievements should be immediately visible to the player. However, you might hide an achievement if the achievement is intended to be a surprise or if it is available only when the player purchases additional content.

  Your application reveals a hidden achievement by reporting progress towards completing the achievement.

You edit all the data for your achievements in iTunes Connect. You also localize descriptions and titles there for the languages you intend to support.

Each game owns its achievement descriptions; you may not share achievement descriptions between multiple games.

For details on configuring achievement descriptions for your application, see iTunes Connect Developer Guide.

Reporting Achievement Progress to Game Center

When the player makes progress towards completing an achievement in your game, your game reports this progress to Game Center. By storing progress information in Game Center, you allow the Game Center app to display the player's achievements. Progress on an achievement is always reported for the local player.

Your application reports the player's progress by setting a floating-point percentage, from `0.0` to `100.0` that represents how much of the achievement the player has completed. It is up to you to decide how that percentage is calculated and when it changes. For example, if your achievement was "Find ten gold coins", then you might increment the percentage by 10% each time the player finds a coin. On the other hand, if the player earns an achievement simply for discovering a location in your game, then you might simply report the achievement as 100% complete the first time you report any progress. If your application does support incremental progress for an achievement, it should report that progress to Game Center whenever the value changes.

When you report progress to Game Center, two things happen:

- If the achievement was previously hidden, it is revealed to the player. The achievement is revealed even if your player has made no progress on the achievement (a percentage of `0.0`).
- If the reported value is higher than the previous value reported for the achievement, the value on Game Center is updated to the new value. Players never lose progress on achievements.

When the progress reaches 100 percent, the achievement is marked completed, and both the image and completed description appear when the player views the achievements screen.

Your application reports progress to Game Center using a `GKAchievement` object. Listing 4-1 shows how to report progress to Game Center. First, a new achievement object is initialized using an identifier string for an achievement your application supports. Next, the object's `percentComplete` property is changed to reflect the player's progress. Finally, the object's `reportAchievementWithCompletionHandler:` method is called, passing in a block to be notified when the report is sent.

Listing 4-1 Reporting progress on an achievement

```
- (void) reportAchievementIdentifier: (NSString*) identifier percentComplete:
(float) percent

{

    GKAchievement *achievement = [[[GKAchievement alloc] initWithIdentifier:
identifier] autorelease];
```

```
    if (achievement)
    {
         achievement.percentComplete = percent;
         [achievement reportAchievementWithCompletionHandler:^(NSError *error)
             {
                  if (error != nil)
                  {
                      // Retain the achievement object and try again later (not
shown).
                  }
             }];
    }
}
```

Your application must handle errors when it fails to report progress to Game Center. For example, the device may not have had a network when you attempted to report the progress. The proper way for your application to handle network errors is to retain the achievement object (possibly adding it to an array). Then, your application needs to periodically attempt to report the progress until it is successfully reported. The `GKAchievement` class supports the `NSCoding` protocol to allow your application to archive an achievement object when it moves into the background.

Loading Achievement Progress

In order to display a player's progress towards achievements, your application retrieves the player's progress information from Game Center by calling the `loadAchievementsWithCompletionHandler:` class method. If the operation completes successfully, it returns an array of `GKAchievement` objects, one object for each achievement your application previously reported progress for.

A logical time to load the player's progress is immediately after the player is authenticated. Listing 4-2 (page 41) shows how your application might implement this.

Listing 4-2 Loading achievement progress in the authentication block handler

```
- (void) authenticatePlayer
{
    [[GKLocalPlayer localPlayer] authenticateWithCompletionHandler:^(NSError *error)
  {
        if (error == nil)
        {
            [self loadAchievements];
            // Perform other authentication-completed tasks here.
        }
     }];
}

- (void) loadAchievements
{    [GKAchievement loadAchievementsWithCompletionHandler:^(NSArray *achievements,
 NSError *error) {
        if (error != nil)
        {
            // handle errors
        }
        if (achievements != nil)
        {
            // process the array of achievements.
        }
       }];
}
```

As the player makes progress through your game, you want to update their progress on Game Center. If your game has previously reported progress towards this achievement, your game should first load the player's progress from Game Center and update that achievement object. If the player made progress on an achievement that the player has never made progress on before, your application should create a new achievement object. An easy way to manage these achievement objects in your game is by using a mutable dictionary, using the `identifier` property as a dictionary key, and the achievement object as the contents for that key. Here's how to modify Listing 4-1 (page 40) and Listing 4-2 (page 41) to use a dictionary:

1. Add a mutable dictionary property to your class that report achievements. This dictionary stores the collection of achievement objects.

   ```
   @property(nonatomic, retain) NSMutableDictionary *achievementsDictionary;
   ```

2. Initialize the achievements dictionary.

```
achievementsDictionary = [[NSMutableDictionary alloc] init];
```

3. Modify your code that loads loads achievement data to add the achievement objects to the dictionary.

```
- (void) loadAchievements
{
    [GKAchievement loadAchievementsWithCompletionHandler:^(NSArray
*achievements, NSError *error)
        {
            if (error == nil)
            {
                for (GKAchievement* achievement in achievements)
                   [achievementsDictionary setObject: achievement forKey:
 achievement.identifier];
            }
        }];
}
```

4. Implement a method that tests the dictionary for a particular achievement identifier. If the identifier does not exist as a key in the dictionary, create a new achievement object and add it to the dictionary:

```
- (GKAchievement*) getAchievementForIdentifier: (NSString*) identifier
{
    GKAchievement *achievement = [achievementsDictionary
objectForKey:identifier];
    if (achievement == nil)
    {
        achievement = [[[GKAchievement alloc]
initWithIdentifier:identifier] autorelease];
        [achievementsDictionary setObject:achievement
forKey:achievement.identifier];
    }
    return [[achievement retain] autorelease];
}
```

5. Modify the code in Listing 4-1 (page 40) to call `getAchievementForIdentifier:` to retrieve the achievement object.

```
- (void) reportAchievementIdentifier: (NSString*) identifier
percentComplete: (float) percent

{

    GKAchievement *achievement = [self
getAchievementForIdentifier:identifier];

    if (achievement)

    {

         achievement.percentComplete = percent;

         [achievement reportAchievementWithCompletionHandler:^(NSError
*error)

             {

                  if (error != nil)

                  {

                     // Retain the achievement object and try again later
 (not shown).

                  }

             }];

    }

}
```

Resetting Achievement Progress

You may want to allow the player to reset their progress on achievements in your application. To allow this, your application calls the `resetAchievementsWithCompletionHandler:` class method. Listing 4-3 (page 44) demonstrates how to do this. First, this method clears any locally cached achievement objects created by the previous example. Then, it erases the player's progress stored on Game Center.

Listing 4-3 Resetting achievement progress

```
- (void) resetAchievements

{

// Clear all locally saved achievement objects.

    achievementsDictionary = [[NSMutableDictionary alloc] init];

// Clear all progress saved on Game Center

[GKAchievement resetAchievementsWithCompletionHandler:^(NSError *error)
```

```
    {
        if (error != nil)
            // handle errors
    }];
}
```

When your application resets a player's progress on achievements, all progress information is lost. Hidden achievements, if previously shown, are hidden again until your application reports progress on them. For example, if the only reason those achievements were originally hidden was because they were associated with in-app purchases, then you would reveal those achievements again.

Displaying Achievements Using an Achievement View Controller

The Game Center application provides a screen that displays the achievements for your application. You should also add an achievements view in your application. The `GKAchievementViewController` class provides a standard interface screen for viewing your game's achievements. To use an achievement view controller, present the achievement view controller modally using another view controller created by your application.

Listing 4-4 (page 45) shows how to have your view controller present an achievements screen. This method creates a new achievement view controller, sets the achievement delegate to point to itself, and then presents the screen.

Listing 4-4 Presenting the standard achievement view to the player

```
- (void) showAchievements
{
    GKAchievementViewController *achievements = [[GKAchievementViewController
alloc] init];
    if (achievements != nil)
    {
        achievements.achievementDelegate = self;
        [self presentModalViewController: achievements animated: YES];
    }
    [achievements release];
}
```

Your view controller must also implement the `GKAchievementViewControllerDelegate` protocol so that it can be notified when the user dismisses the achievements screen. Listing 4-5 shows an implementation of the delegate that removes the modal view from the screen.

Listing 4-5 Responding when the player dismisses the achievements view

```
- (void)achievementViewControllerDidFinish:(GKAchievementViewController
*)viewController
{
    [self dismissModalViewControllerAnimated:YES];
}
```

Creating a Custom Achievement User Interface

If you want to create your own custom achievements user interface, you can load achievement descriptions from Game Center and combine those descriptions with the user's progress. Game Kit provides achievement descriptions to your game using the `GKAchievementDescription` class. A `GKAchievementDescription` object's properties match the information you added in iTunes Connect for an achievement. This section describes how your application loads those descriptions from Game Center.

Loading achievement descriptions is a two-step process. First, your application loads the text descriptions for all achievements in your game. Next, for each achievement completed, your application loads the completed achievement image. This second step allows your application to load only images you need, which reduces your application's memory footprint.

Listing 4-6 shows how to load the achievement descriptions. The code does this by calling the `loadAchievementDescriptionsWithCompletionHandler:` class method.

Listing 4-6 Retrieving achievement metadata from Game Center

```
- (void) retrieveAchievmentMetadata
{
    [GKAchievementDescription loadAchievementDescriptionsWithCompletionHandler:
        ^(NSArray *descriptions, NSError *error) {
            if (error != nil)
                // process the errors
            if (descriptions != nil)
                // use the achievement descriptions.
        }];
}
```

Although the properties are self-explanatory, one critical property worth noting is the `identifier` property. This corresponds to the identifier string used on a `GKAchievement` object. When you design your custom user interface, you use the `identifier` property to match each `GKAchievementDescription` object to the corresponding `GKAchievement`object that records the player's progress on that achievement.

The value of the `image` property is `nil` until you tell the object to load its image data. Listing 4-7 shows how your application tells an achievement description to load the image.

Listing 4-7 Loading a completed achievement image

```
[achievementDescription loadImageWithCompletionHandler:^(UIImage *image, NSError
*error) {
    if (error == nil)
    {
        // use the loaded image. The image property is also populated with the
same image.
    }
  }];
```

The `GKAchievementDescription` class provides two default images your application can use. The `incompleteAchievementImage` class method returns an image that should be used for any achievement that has not been completed. If your application is unable to load an image for a completed achievement (or wants to display an image while the custom image is loading), the `placeholderCompletedAchievementImage` class method provides a generic image for a completed achievement.

Multiplayer

Multiplayer matchmaking is a service that can be used by any game that works with Game Center. Matchmaking makes it easy for like-minded players playing your game to discover each other and play a game together.

Checklist for Adding Matchmaking to Your Game

To add matchmaking to your application, you have some decisions to make about your game. How many players can play at once? What does a network version of your game look like? These design decisions impact the code you need to write to implement matchmaking using Game Center. When you are ready to add matchmaking to your application, use this checklist to guide you through the process:

- If you haven't already done so, add code to your application to authenticate the local player. See "Working with Players in Game Center" (page 20).
- Add code to display the matchmaking screen to the user. In the code, you create a match request and use it to initialize a matchmaker view controller. See "Creating a New Match Starts with a Match Request" (page 49) to learn about match requests and "Presenting the Matchmaking User Interface" (page 50) to learn how to display the view controller..
- Add code that processes invitations. Game Kit calls your invitation handler whenever it has a pending invitation for the local player. See "Processing Invitations from Other Players" (page 52).
- Optionally, add code to programmatically find matches. See "Finding a Match Programmatically" (page 54).
- Optionally, add advanced matchmaker functionality to your game. Read "Advanced Matchmaking Topics" (page 55) for a list of functionality you might want to add. For example, if your game offers multiple modes in your game, you can restrict matchmaking so that players find only those players interested in the same game mode.

Understanding Common Matchmaking Scenarios

Game Center offers a few different ways that players can be grouped into a match.

The first and most common scenario is that a player is already playing your game, and wants to create a multiplayer match with one or more friends. Your game modally presents the standard matchmaking view. The player uses this screen to invite friends into the match. When the hosting player invites a friend, that friend receives a push notification asking them if they want to join the game. When a friend accepts the invitation, the device automatically launches your game (or moves it to the foreground if it was already running). Once all the friends have your game launched, they are notified that the match is ready. When the players start the match, each copy of your game dismisses the matchmaking screen and then uses the match to start playing the game.

A player can also create a network match using the Game Center app. When they invite a friend into a multiplayer game, your application is launched on both devices, and each copy of your application receives an invitation to join the game. A key advantage to this scenario is that your application does not have to do any additional work to support it — if your application already supports invitations, you get this for free!

Whether the player creates a match using your application or the game center application, the standard matchmaking screen allows the hosting player to fill empty game slots with other Game Center players that were already waiting for a match. For example, if your game requires four players, a group of three friends can fill the fourth slot without any additional work on your part.

Your game can provide automatic matching to create a match. When you use automatic matching, players do not invite other players into the match. Instead, you send a request to Game Center and the player is connected into a match with any other players waiting to join a match (not just friends). Automatic matching does not show any user interface; you can customize what the player sees while the match is being formed.

Note Matchmaking can be done only with other copies of the same application (that is, applications that share the same bundle identifier). You cannot perform matchmaking between two different applications.

Creating a New Match Starts with a Match Request

All new matches start with a match request, a `GKMatchRequest` object. The match request describes the desired parameters for the match.

Game Center uses the match request to customize its matchmaking behavior. For example, when your application displays the standard matchmaking interface, the request is used to customize the screen's appearance. If, on the other hand, your application uses automatic matching, the match request is used to find compatible players to gather into a new match.

Listing 5-1 shows the smallest amount of code that can create a new match request. All match requests must set the minimum and maximum number of players allowed in the match; these settings ensure that Game Center always matches the correct number of players for the match.

Listing 5-1 Creating a match request

```
GKMatchRequest *request = [[[GKMatchRequest alloc] init] autorelease];
request.minPlayers = 2;
request.maxPlayers = 2;
```

Note As of iOS 4.2, `minPlayers` must be at least 2 and `maxPlayers` must be equal to or less than 4. *Hosted* matches are an advanced topic, and allow up to a maximum of 16 players. See "Hosting Games on Your Own Server" (page 60) for more information.

Presenting the Matchmaking User Interface

When your application creates a match, your application's view controller presents the standard matchmaking interface screen modally to the player. That player is then free to invite friends into the match, or choose to fill empty slots through automatic matching.

Listing 5-2 shows how your view controller presents the matchmaking screen. This code creates and configures a match request, and uses it to initialize a new matchmaker view controller object. Your view controller sets itself as the matchmaker view controller's delegate and then presents the matchmaking screen modally to the player. The player interacts with the matchmaking screen, inviting or matching other players and eventually starting the game.

Listing 5-2 Creating a new match

```
- (IBAction)hostMatch: (id) sender
{
    GKMatchRequest *request = [[[GKMatchRequest alloc] init] autorelease];
    request.minPlayers = 2;
    request.maxPlayers = 2;

    GKMatchmakerViewController *mmvc = [[[GKMatchmakerViewController alloc]
initWithMatchRequest:request] autorelease];
    mmvc.matchmakerDelegate = self;
    [self presentModalViewController:mmvc animated:YES];
}
```

Your delegate must implement a few methods to respond to events. Each of these methods should dismiss the view controller, then perform any application-specific actions required for your application.

The `matchmakerViewControllerWasCancelled:` delegate method is called when the player dismisses the matchmaking screen by tapping the cancel button before a match has been created. In most cases, you should simply return to the previous screen in your game.

Listing 5-3 Implementing the cancellation method

```
- (void)matchmakerViewControllerWasCancelled:(GKMatchmakerViewController
*)viewController
{
    [self dismissModalViewControllerAnimated:YES];
    // implement any specific code in your application here.
}
```

The `matchmakerViewController:didFailWithError:` delegate method is called when matchmaking encounters an error while trying to set up the match. For example, the device may have lost its network connection. Your implementation of this method should read the `error` property and display a message to the user, then return to your game's previous screen.

Listing 5-4 Implementing the error handling method

```
- (void)matchmakerViewController:(GKMatchmakerViewController *)viewController
didFailWithError:(NSError *)error
{
    [self dismissModalViewControllerAnimated:YES];
    // Display the error to the user.
}
```

Finally, when the match has been created and everyone is ready to start, your delegate receives a call to let you know the match was successfully created. Game Kit provides a `GKMatch` object to your game. Listing 5-5 (page 52) provides a method you might implement in your own class to receive the match object. This method assigns the match object to a retaining property and adds a delegate to the match. If all the players are already connected to the match, it begins exchanging data with the other participants. Otherwise, it waits until the remaining players to connect to the match; in that case the match delegate is responsible for starting the match. For more information on how to implement the match delegate and how to use the match to exchange information with other participants, see "Using Matches to Implement Your Network Game" (page 63).

Listing 5-5 Implementing the match found method

```
- (void)matchmakerViewController:(GKMatchmakerViewController *)viewController
didFindMatch:(GKMatch *)match
{
    [self dismissModalViewControllerAnimated:YES];
    self.myMatch = match; // Use a retaining property to retain the match.
    match.delegate = self;
    if (!self.matchStarted && match.expectedPlayerCount == 0)
    {
       self.matchStarted = YES;
       // Insert application-specific code to begin the match.
    }
}
```

Processing Invitations from Other Players

When a player invites a friend to join a match, the friend's device displays a push notification. If the friend accepts the invitation, your application is automatically launched on the friend's device so that the friend can be connected into the match. Your application receives this invitation by implementing an invitation handler.

An invitation handler creates a match view controller and initializes it with data provided to it by Game Kit. It performs activities similar to those found in Listing 5-2 (page 50) — in most cases, you can reuse the same delegate code. The invitation handler takes two different parameters; on any call to your invitation hander, only one of these parameters holds a non-`nil` value:

- The `acceptedInvite` parameter is non-`nil` when your game receives an invitation directly from another player. In this situation, the other player's game has already created the match request, so your application running on the invitee's device does not need to create a match request.
- The `playersToInvite` parameter is non-`nil` when your game is launched directly from the Game Center app to host a match. This parameter holds an array of player identifiers listing the players your game should invite into the match. Your game must create a new match request, assign its parameters as you

would normally, and then set the match request's `playersToInvite` property to the value passed in the `playersToInvite` parameter. When the matchmaking screen is displayed, it is pre-populated with the list of players already included in the match.

Important Your application should provide an invitation handler as early as possible after your application authenticates the local player; an appropriate place to set the handler is in the completion handler that executes after the local player is authenticated. It is critical that your application authenticate the local player and set the invitation handler as soon as possible after launching, specifically so that you can handle the invitation that caused your application to be launched.

Listing 5-6 shows a typical invitation handler. Because the player has already accepted the invitation, the game stops any gameplay already in progress, instantiates a new view controller object using the invitation handler's parameters and presents it to the player.

Listing 5-6 Adding an invitation handler

```
[GKMatchmaker sharedMatchmaker].inviteHandler = ^(GKInvite *acceptedInvite, NSArray
 *playersToInvite) {
   // Insert application-specific code here to clean up any games in progress.
   if (acceptedInvite)
    {
        GKMatchmakerViewController *mmvc = [[[GKMatchmakerViewController alloc]
initWithInvite:acceptedInvite] autorelease];
        mmvc.matchmakerDelegate = self;
        [self presentModalViewController:mmvc animated:YES];
    }
    else if (playersToInvite)
    {
        GKMatchRequest *request = [[[GKMatchRequest alloc] init] autorelease];
        request.minPlayers = 2;
        request.maxPlayers = 4;
        request.playersToInvite = playersToInvite;

        GKMatchmakerViewController *mmvc = [[[GKMatchmakerViewController alloc]
initWithMatchRequest:request] autorelease];
        mmvc.matchmakerDelegate = self;
        [self presentModalViewController:mmvc animated:YES];
    }
};
```

Finding a Match Programmatically

Your application can also ask Game Center to find a match for the player without presenting a user interface. For example, your application could offer an "instant match" button to let someone request a new match by pressing a single button.

Listing 5-7 shows how to programmatically request a match. The code first creates a match request, then retrieves the matchmaker singleton object and asks for a match. The provided completion handler is called when a match is found or if an error occurs. If a match was returned to the completion handler, the match is assigned to a retaining property and used to start the multiplayer match.

Listing 5-7 Programmatically finding a match

```
- (IBAction)findProgrammaticMatch: (id) sender
{
    GKMatchRequest *request = [[[GKMatchRequest alloc] init] autorelease];
    request.minPlayers = 2;
    request.maxPlayers = 4;

    [[GKMatchmaker sharedMatchmaker] findMatchForRequest:request
withCompletionHandler:^(GKMatch *match, NSError *error) {
        if (error)
        {
            // Process the error.
        }
        else if (match != nil)
        {
            self.myMatch = match; // Use a retaining property to retain the match.
            match.delegate = self;
            if (!self.matchStarted && match.expectedPlayerCount == 0)
            {
               self.matchStarted = YES;
               // Insert application-specific code to begin the match.
            }
        }
    }];
}
```

Adding Players to an Existing Match

Sometimes you may already have a match, and just want to add players to it. For example, if your game requires four players and a player gets disconnected, you might want to offer the option to find a replacement, instead of aborting the match in progress.

To do this, you use code similar to that found in Listing 5-7 (page 54), but instead of calling the `findMatchForRequest:withCompletionHandler:`, your application calls the `addPlayersToMatch:matchRequest:completionHandler:` method, adding the match to add the players to as an additional parameter.

Canceling a Match Request

If your app includes support for programmatic matching, you should provide a user interface that allows the player to cancel the match request. Listing 5-8 shows how your game can terminate a pending search.

Listing 5-8 Canceling a match search

```
[[GKMatchmaker sharedMatchmaker] cancel];
```

Advanced Matchmaking Topics

Once you have matchmaking working in your game, consider adding one or more of the advanced matchmaking features into your application.

- **Player groups** allow you to create subsets of players. When your app uses a player group, only players in the same group may be automatically matched with each other. See "Player Groups" (page 56).
- **Player attributes** allow you to define specific roles that a player can fill in your game. When your match request includes a player role, automatic matching places them only in a match that requires a player to fill that role. See "Player Attributes" (page 57).
- **Player activity** allows your game to query Game Center for a rough approximation of how many players are playing your game online. This allows you to tell the player whether they will quickly find a match. See "Searching for Player Activity" (page 59).

- **Hosted matches** allow your game to use Game Center's matchmaking service to find players, but use your own server to connect the players to each other. Hosted matches require you to write custom networking code but offer larger player groups and the ability to add your own server into the match. See "Checklist for Adding Matchmaking to Your Game" (page 48)

Each of these advanced features can be used individually, or combined in the same application:

Player Groups

Game Center's default behavior is to automatically match any player waiting to play your game with any match that needs more players. While this has the advantage of matching players quickly into matches, it may not be the behavior you want. Your application might want to only match like-minded players with each other. Therefore, you might want to group players into smaller groups. For example, you might want to do one of the following:

- Separate players by player skill level.
- Separate players by the set of rules used to adjudicate the match.
- Separate players based on the map the match is played on.

You implement this separation of players in your application by using player groups. With player groups, you can help players get matched directly into the kinds of matches they want to play.

Implementing Player Groups

A player group is represented by the `playerGroup` property on a match request. When the value stored in the `playerGroup` property is non-zero, the player is only matched with players whose match requests specified the same 32-bit value. When you design your game, you decide what values to provide and when they are provided; for example, your game might present its own user interface to allow the player to select the parameters used to define the style of play they are interested. Each option the player chooses may contribute a portion of the value stored in the `playerGroup` property.

Listing 5-9 initializes a match request and adds a player group. In this example, the player group value is calculated as by performing an OR operation between two constants, one representing the map the match is played on, and the other representing the rules used to play the match.

Listing 5-9 Creating a player group based on the map and rule set

```
GKMatchRequest *request = [[[GKMatchRequest alloc] init] autorelease];
request.minPlayers = 2;
request.maxPlayers = 4;
request.playerGroup = MyMap_Forest | MyRulesCaptureTheFlag;
```

Player Attributes

Some games offer different roles that a player can play in a multiplayer match. For example:

- Many role playing games offer different character roles (sometimes known as classes). Each role defines different strengths, weaknesses and abilities for the player to bring to the match.
- Similarly, a sports game might think of roles as a particular position the player plays on the field, such as a quarterback or a linebacker.

Using player attributes, you can add support to your game to allow a player to choose a specific role they want to play once the match starts. Game Center automatically matches the player into a match that requires a player to fill that role.

Player attributes have some limitations:

- Only one player can fill each role.
- Your game defines a complete set of roles; all roles defined by your game must be filled.
- Each match request can only request a single role.
- Roles are checked only for auto-matched players. If the player invites friends to join the match, friends do not get to pick a role.
- Roles are not displayed in the standard user interface for matchmaking provided by Game Kit. Your application must provide its own custom user interface to allow players to choose a role.
- The match object returned to your application does not tell you which roles the players selected. Your game must send the role-selection information separately after the match is created.

Implementing Player Attributes

Player attributes are implemented using the `playerAttributes` property on the match request. By default, the value of this property is `0`, which means that the attributes property is ignored. If the value is non-zero, Game Center uses it to match a specific role.

To create the roles a player can play in your game, you create a 32-bit mask for each role a player can fill in your game. It is up to you to add a custom user interface in your game that allows the player to select a role. You set the `playerAttributes` property on the match request to the mask for the player’s selection. Then, show the matchmaking interface or perform automated matching, as normal.

Listing 5-10 Setting the player attributes on the match request

```
GKMatchRequest *request = [[[GKMatchRequest alloc] init] autorelease];
request.minPlayers = 4;
request.maxPlayers = 4;
request.playerAttributes = MyRole_Wizard;
```

When Game Center uses automatic matching to find players for the match, it only adds players to the match when those player's masks do not overlap the masks of any players already in the match. The algorithm looks roughly like this:

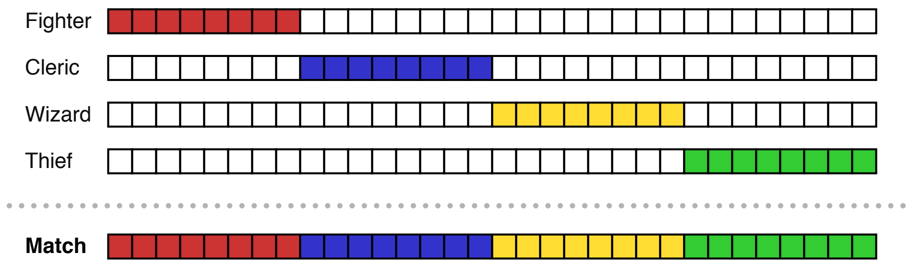
1. A match's mask starts with the mask of the inviting player.
2. Game Center looks for players with match requests that have a non-zero player attributes value. Game Center adds a player to the match only if no bits in the the requesting player's attributes mask overlap with bits present in the match's current mask. That is, if you used a logical AND operation between the match's mask and the player's mask, the result would be `00000000h`.
3. After adding a player to the match, the value of the new player's player attributes value is added to the match's mask.
4. If the match's mask equals `FFFFFFFFh`, then the match is considered complete. Otherwise, it looks for another player.

An example might help. Assume that you are creating a role playing game with four roles: Fighter, Wizard, Thief, and Cleric. Every match consists of four players, the match must have one and only one player for each role. To implement this requirement, your application creates masks for each role that match the four-step algorithm above. Listing 5-11 (page 58) provides an example set of masks.

Listing 5-11 Creating the masks for the character classes

```
#define MyRole_Fighter 0xFF000000
#define MyRole_Cleric 0x00FF0000
#define MyRole_Wizard 0x0000FF00
```

```
#define MyRole_Thief 0x000000FF
```



None of the role masks overlap; if any two masks are joined by AND, the resulting value is always `00000000h`. When all four role masks are added together, the entire match mask is filled (`FFFFFFFFh`).

Searching for Player Activity

Players who are looking for a multiplayer match often want to be matched immediately, or at least be aware of when matchmaking may take longer. For example, if a player is on during a period of time when players are not regularly online, the number of players who are interested in joining a match may be substantially lower than during prime-time. The determination of how many players are online is referred to as **player activity**. The `GKMatchmaker` class provides a pair of methods you can use to test for the activity on Game Center related to your application.

Listing 5-12 Searching for all activity for your application on Game Center

```
- (void)findAllActivity
{
    [[GKMatchmaker sharedMatchmaker] queryActivityWithCompletionHandler:^(NSInteger
 activity, NSError *error) {
        if (error)
        {
            // Process the error.
        }
        else
        {
            // Use the activity value to display activity to the player.
        }
    }];
}
```

If your application uses player groups, you can use the `queryPlayerGroupActivity:withCompletionHandler:` method to retrieve the activity for a specific player group.

The value returned by either method is the number of players who have recently requested a match.

Hosting Games on Your Own Server

By default, you use Game Kit to create a match; Game Kit returns an instance of the `GKMatch` object on each device connected to the match. Each device uses its match object to communicate with other participants. For most games, this default behavior is exactly what you want. The `GKMatch` class provides an abstraction for the underlying network topology so that your application can simply focus on sending data to other participants without worrying about how the data is transmitted. However, some games may want to support more players than the maximum number of supported players on Game Center or may want their own server to arbitrate the match. In these cases, you can use matchmaking to find players for a **hosted match**. A hosted match does not create `GKMatch` objects. Instead, each copy of your application receives the player identifiers for all of the players in the match. Your application must take additional steps to connect the players to your server.

Creating a hosted match requires your application to implement all of the low-level networking required for your application. In particular, you must do all of the following in your application:

- You must design and implement your own networking code to connect each device to your server.
- You must design and implement your own networking protocol to inform other devices of the state of any participant in the match.
- If your application uses Game Kit's standard matchmaking user interface, you must make sure each device informs Game Kit after it connects to your server. This information allows Game Kit to update its user interface.
- You must design and implement your own server implementation to use a player's player identifier string to route data to his or her device.
- Because you are not using a `GKMatch` object object, voice chat is not provided. However, the `GKVoiceChatService` class can be used to implement voice chat over the network connection you provided. See "In-Game Voice" (page 88).

Note A hosted match can support a minimum of 2 and a maximum of 16 players.

The rest of this section describes how to alter your matchmaking code to create a hosted match.

Whether your application creates a hosted match using the standard matchmaking user interface or it creates a hosted match programmatically, the behavior is similar. Instead of receiving an initialized `GKMatch` object, your application instead receives a list of player identifiers for the players to be connected into the match. Each game client must establish a connection to your server, and transfer the player identifier for the authenticated local player, as well as the list of players for the match, to your server. Your server then takes the list of player identifiers and performs whatever game logic is necessary to network the participants together.

Creating a Hosted Match through the View Controller

To create a hosted match through the view controller, you set the view controller's `hosted` property to yes before presenting the matchmaking screen to the player. Listing 5-13 modifies the method provided in Listing 5-2 (page 50)

Listing 5-13 Creating a hosted match

```
- (IBAction)hostMatch: (id) sender
{
    GKMatchRequest *request = [[[GKMatchRequest alloc] init] autorelease];
    request.minPlayers = 2;
    request.maxPlayers = 2;

    GKMatchmakerViewController *mmvc = [[[GKMatchmakerViewController alloc]
initWithMatchRequest:request] autorelease];
    mmvc.matchmakerDelegate = self;
    mmvc.hosted = YES;

    [self presentModalViewController:mmvc animated:YES];
}
```

Similarly, when your invitation handler receives an invitation, the `GKInvite` object's `isHosted` property states whether the match was created as a hosted match. Your invitation handler should set the `hosted` property of the matchmaker view controller appropriately.

In addition to displaying the user interface, each device must also connect to your server. When the device connects to the server, your server should inform all devices already connected to the hosted match. Each device must then call its view controller's `setHostedPlayerReady:` method, passing the player identifier for the player that just connected. This allows the matchmaking screens on all the participants' devices to be updated.

Once all the participants are connected to the server, and the players are ready, your delegate object is called to start the game. In standard Game Center matchmaking, your application implemented the `matchmakerViewController:didFindMatch:` method to receive the completed match object. For a hosted game, your application implements the `matchmakerViewController:didFindPlayers:` method instead.

Creating a Hosted Match Programmatically

Creating a hosted match programmatically is almost identical to creating regular matches. You create a match request, get the shared matchmaker singleton, and use it to find the match. To create a hosted match, call the matchmaker's `findPlayersForHostedMatchRequest:withCompletionHandler:` method instead. When you call this method, your application receives an array of player identifiers for the players in the match.

Using Matches to Implement Your Network Game

A match is group of players whose devices are connected to each other over a network by Game Center. Matches allow data and voice to be transmitted to other participants in the match. Game Center manages the difficult effort of finding other players and establishing the network between them. This frees you to work on designing your network game.

Checklist for Working with Matches

Follow these steps to implement the networking code for your game:

- If you haven't already done so, write code to use Game Center's matchmaking service to create a match. The matchmaking service establishes a connection between the players and returns a `GKMatch` object to your application. Your application then interacts with this match object to send data to other participants. See "Multiplayer" (page 48) for more information on creating matches.
- Design the network messages your game uses to communicate with other participants in the match. Most games exchange information at the start of the match to provide each participant the same initial game state, then send updates as events occur within the game. See "Designing Your Network Game" (page 64).
- Write code to use the match object to send data to other match participants.See "Sending Data to Other Players" (page 67).
- Implement a match delegate to handle all the necessary events that can occur during the match:
  - The delegate is notified when other players are connected at the start of the game. Your delegate must wait until everyone is connected before starting the game. See "Starting the Match" (page 66).
  - The delegate receives the data that other players send. See "Receiving Data from Other Players" (page 68).
  - If players get disconnected while your game is running, the delegate receives a notification and must decide whether to discontinue the match or reconfigure your game to handle the reduced number of players. See "Disconnecting from a Match" (page 68).
- Optionally, add support for voice between the match participants. See "Adding Voice Chat to a Match" (page 69).

Designing Your Network Game

Each instance of your game relies on a `GKMatch` object to exchange data with the other instances connected to the match. The `GKMatch` class does not define the format or content of your network messages. Instead, it simply sees your messages as bytes to transmit. This gives you great flexibility in designing your network game. The rest of this section describes key concepts you should understand before implementing your network game.

Whenever you send data to other participants, you decide how much effort the match should use to send the data. Matches can send your data **reliably**, where the match retransmits the data until it is received by the target(s), or **unreliably**, where it sends the data only once.

- A reliable transmission is simpler, but potentially slower; a slow or error-prone network may require the device to send the message multiple times before it is successfully delivered to its intended recipients. A match also guarantees that multiple reliable messages sent from one device to the same recipient are delivered in the order they were sent.
- Messages transmitted unreliably may never reach their destination or may be delivered out of order. Unreliable transmissions are most useful for real-time transactions where any delay in transmission invalidates the contents of the message. For example, if your game transmits position and velocity information for a dead-reckoning algorithm, a delayed transmission could mean position data that is badly out of date by the time it is delivered to the recipient. Instead, by using unreliable messages, your game instead sends new messages with updated position and reckoning information.

The size of your messages also plays an important role in how quickly the data can be delivered to its targets. Large messages must be split into smaller packets (such splitting is called **fragmentation**) and reassembled by each target. Each of these smaller packets might be lost during transmission or delivered out of order. Large messages should be sent reliably, so that Game Kit can handle the fragmentation and assembly. However, the process of resending and assembly takes time. You should not use reliable transmissions to send large amounts of real-time data.

Be mindful that your network data is being transmitted across servers and routers that are out of your control. Your messages are subject to inspection and modification by other devices on the network. When your application on one device receives network data from participants on other devices, it should treat that message as *untrusted* data, and validate its contents before using it. See *Secure Coding Guide* for information on how to avoid security vulnerabilities.

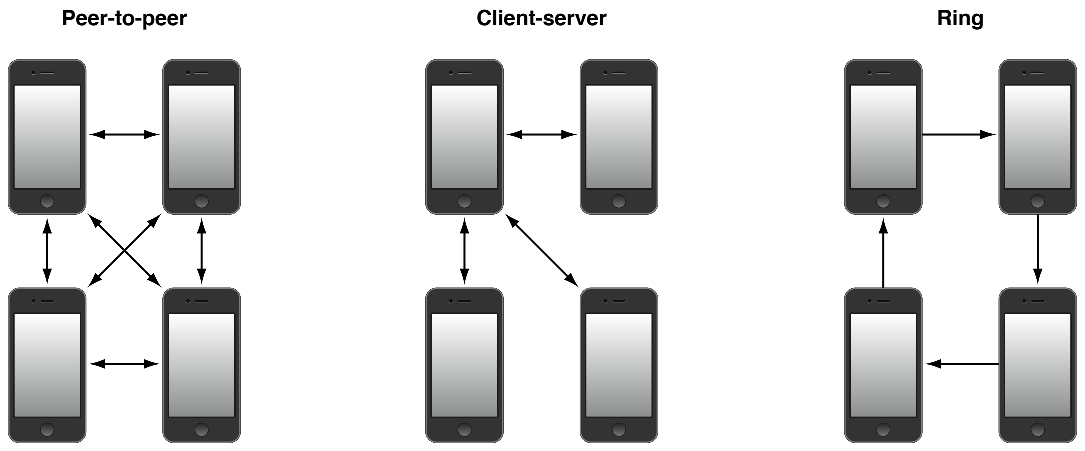
Here are some general guidelines to follow when designing your game's networking:

- Your message format should include a way to differentiate between different kinds of messages. The `GKMatch` class does not know anything about the contents of your messages, so you must implement that functionality in your application. For example, you might create an enumerated type that identifies different kinds of messages, and start each message with that enumerated type.

- Send messages at the lowest frequency that allows your game to function well. Your game's graphics engine may be running at 30 to 60 frames per second, but your networking code can send updates much less frequently.
- Use the smallest message format that gets the job done. Messages that are sent frequently or messages that must be received quickly by other participants should be carefully scrutinized to ensure that no unnecessary data is being sent.
- Pack your data into the smallest representation you can without losing valuable information. For example, an integer in your program may use 32 or 64 bits to store its data. If the value stored in the integer is always in the range `1` through `10`, you can store it in your network message in only four bits.
- Limit the size of unreliable messages to 1000 bytes or smaller in size.
- Limit the size of reliable messages to 87 kilobytes or smaller in size.
- Send messages only to the participants that need the information contained in the message. For example, if your game has two different teams, team-related messages should be sent only to the members of the same team. Sending data to all participants in the match uses up networking bandwidth for little gain.

  Although the `GKMatch` object creates a full peer-to-peer connection between all the participants, you can reduce the network traffic by layering a ring or client-server networking architecture on top of it. Figure 6-1 (page 65) shows three possible network topologies for a four-player game. On the left, a peer-to-peer game has 12 connections between the various devices. However, you could layer a client-server architecture on top of this by nominating one of the devices to act as the host. If your application transmits to or from the host only, you can halve the number of connections. A ring architecture allows devices to forward network packets to the next device only, but further reduces the number of connections. Each topology provides different performance characteristics, so you may need to test to find a model that provides the performance your application requires.

Figure 6-1 Network topologies



- Specify how to handle network disruptions. Networks are an inherently unreliable medium of communication. A participant can be disconnected at any time while the match is in progress. Your game must handle disconnection messages. For example, if you implemented your game to use a client-server topology, then when the server disconnects from the match, your game might want to nominate a new device to become the new server.

Starting the Match

When Game Kit delivers a `GKMatch` object to your game, the connections to other participants in the match may not be established yet. The `playerIDs` array on that user's device can be empty if no other players are connected, or it might hold a subset of players that have already been connected. Your game must wait until all players are connected before starting the match. To determine how many players are waiting to join the match, your game reads the match's `expectedPlayerCount` property. When the value of this property reaches `0`, all players are connected and the match is ready to start.

The appropriate place to perform this check is in the match delegate's `match:player:didChangeState:` method. This method is called whenever a member of the match connects or disconnects. Listing 6-1 (page 66) is an example of an implementation of your `match:player:didChangeState:` method. In this example, the match delegate defines its own `matchStarted` property to record whether the match is already in progress. If the match has not started and the count of expected players reaches zero, the method starts the match. In your game, this is where any initial match state would be transmitted to other players or where additional negotiations between the different participants takes place.

Listing 6-1 Starting a match

```
- (void)match:(GKMatch *)match player:(NSString *)playerID
didChangeState:(GKPlayerConnectionState)state
{
    switch (state)
    {
        case GKPlayerStateConnected:
            // handle a new player connection.
           break;
        case GKPlayerStateDisconnected:
            // a player just disconnected.
           break;
    }
    if (!self.matchStarted && match.expectedPlayerCount == 0)
    {
        self.matchStarted = YES;
        // handle initial match negotiation.
    }
}
```

Sending Data to Other Players

To send data from one device to other devices connected to the match, your game creates a message and encapsulate it in a `NSData` object. You have this message sent to all connected players using the `sendDataToAllPlayers:withDataMode:error:` method, or to a subset of the players using the `sendData:toPlayers:withDataMode:error:` method.

Listing 6-2 shows how an application might send a position update to the other participants. Listing 6-2 fills a structure with position data, wraps it in an `NSData` object, and then calls the match's `sendDataToAllPlayers:withDataMode:error:` method.

Listing 6-2 Sending a position to all players

```
- (void) sendPosition
{
    NSError *error;
    PositionPacket msg;
    msg.messageKind = PositionMessage;
    msg.x = currentPosition.x;
    msg.y = currentPosition.y;
    NSData *packet = [NSData dataWithBytes:&msg length:sizeof(PositionPacket)];
    [match sendDataToAllPlayers: packet withDataMode: GKMatchSendDataUnreliable
error:&error];
    if (error != nil)
    {
        // handle the error
    }
}
```

When the method returns without reporting an error, then the message has been queued and will be sent when the network is available.

Receiving Data from Other Players

When the match receives data sent by another participant, the message is delivered by calling your match delegate's `match:didReceiveData:fromPlayer:` method. Your implementation of this method should decode the message and act on its contents.

```
- (void)match:(GKMatch *)match didReceiveData:(NSData *)data fromPlayer:(NSString
 *)playerID
{
    Packet *p = (Packet*)[data bytes];
    if (p.messageKind == PositionMessage)
        // handle a position message.
}
```

Disconnecting from a Match

When a player is ready to leave a match, your game should call the match object's `disconnect` method. A player may also be automatically disconnected if their device does not respond for a certain period of time. When a player disconnects from the match, the other participants in the match are notified by calling the match delegate's `match:player:didChangeState:` method.

Adding Voice Chat to a Match

If you use a `GKMatch` object to provide your network connection between the players, your game has built-in support for in-game voice. You can create one or more separate voice chat **channels**. Each channel contains a subset of the players that are connected to the match. When a player speaks into a channel, only participants connected to the same channel can hear that player.

Voice chat is available only to participants of the match that have Wi-Fi connections.

Checklist for Adding Voice Chat to a Match

Once your game creates a match, and the players are connected, your game creates and activates one or more voice channels. Follow these steps:

- Decide how many channels your game needs. For example, a free-for-all game might need a single channel for all participants. A game with multiple teams might need a separate channel for each team and an additional channel that includes all players in the game.
- Configure an audio session to enable the microphone. All applications that record or play audio must have an audio session. See "Creating an Audio Session" (page 70).
- Call the match object's `voiceChatWithName:` method to create that voice chat's channel. The match returns a `GKVoiceChat` object that provides the channel. See "Creating Voice Channels" (page 70).
- Call the voice chat object's `start` method when you want to activate the channel. See "Starting and Stopping Voice Chat" (page 71).
- When a player needs to be able to speak into a channel, enable the microphone for that channel. If your application has multiple channels, only one channel may use the microphone at a time. See "Enabling and Disabling the Microphone" (page 71).
- Provide controls in your application to allow the user to enable and disable voice chat. Also provide controls to allow the user to set volume levels or mute players within a channel. See "Controlling the Volume of a Voice Chat" (page 72).
- Decide whether your application should support a push-to-talk model or whether it should continuously sample the microphone. In a push-to-talk application, provide a control the player presses to transmit voice data to other players. In an application that continuously samples the microphone, provide a control to toggle the transmission on and off.

- Optionally, implement an update handler to be called when a player connects or disconnects—or start or stops speaking. For example, you might use an update handler to update your application's user interface so that it changes as players speak. See "Implementing an Update Handler for a Player's State" (page 72).

Creating an Audio Session

Before your application can use the voice chat services provided by the match object, you must create an audio session that has the capability to both play and record sounds. If your game provides other sound effects, you probably already have an audio session. Listing 7-1 (page 70) shows the code necessary to create an audio session that allows the microphone to be used.

Listing 7-1 Setting an audio session that can play and record

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

For more details on creating and using audio sessions, see *Audio Session Programming Guide* .

Creating Voice Channels

Your application never directly creates `GKVoiceChat` objects. Instead, voice chat objects are created on your behalf by the `GKMatch` object. A single match can create multiple channels, and a single player can be assigned to more than one channel at a time. Audio received on any of the channels is mixed together and outputted through the speakers.

In order for multiple participants on different devices to join the same channel, they need a way to identify a particular channel. This identification is accomplished through a **channel name**. A channel name is a string ,defined by your application, that uniquely names the channel. When two or more participants join a channel with the same name, they are automatically connected to a voice chat with each other.

Listing 7-2 provides code that creates two channels. The first call to `voiceChatWithName:` creates a team channel and the second creates a global channel. The code retains both channels.

Listing 7-2 Creating voice channels

```
GKMatch* match;
```

```
GKVoiceChat *teamChannel = [[match voiceChatWithName:@"redTeam"] retain];
GKVoiceChat *allChannel = [[match voiceChatWithName:@"allPlayers"] retain];
```

Starting and Stopping Voice Chat

Voice data is not sent or received through a voice channel until a voice chat is started. You start a voice chat by calling the voice chat object's `start` method:

```
[teamChannel start];
```

After the `start` method is called, the voice chat object on that device connects to other participants in the channel, if the device is on a Wi-Fi network and there is a microphone connected to the device. If either of these conditions is not met, the voice chat object waits until both are true before connecting to the channel.

Similarly, when a player is ready to leave a channel whenever you want a channel to be temporarily turned off, you stop the chat:

```
[teamChannel stop];
```

An advantage to stopping the channel (rather than simply muting the other players) is that the other players are not required to send data to the player who has left the channel. This decrease in data transmission leaves more bandwidth available for your game's messaging.

Enabling and Disabling the Microphone

When you want the player to be able to speak, you enable the microphone. Depending on the nature of your game, you may want to enable the microphone continuously while your game is running, or you may want to include a push-to-talk button in your interface.

A channel enables the microphone by setting the voice chat object's `active` property to `YES`:

```
teamChannel.active = YES;
```

Only one channel can enable the microphone at a time. When you enable the microphone for one channel, the `active` property on the previous owner is automatically set to `NO`.

Controlling the Volume of a Voice Chat

Your game can control the volume of a voice chat in two different ways. First, your game can set the overall volume level for the chat by changing the voice chat object's `volume` property.

```
allChannel.volume = 0.5;
```

The `volume` property accepts values between `0.0` and `1.0`, inclusive. A value of `0.0` mutes the entire channel; a volume of `1.0` outputs voice data at full volume.

Second, your application can selectively mute players in a channel. Typically, if your game intends to mute players, it should offer a user interface that allows the player to choose which players they want to mute. To mute a player, you call the voice chat object's `setMute:forPlayer:` method:

```
[teamChannel setMute: YES forPlayer inPlayerID];
```

To unmute a player, you make the same call, passing `NO` instead.

```
[teamChannel setMute: NO forPlayer inPlayerID];
```

Implementing an Update Handler for a Player's State

You can implement an update handler when your game wants to be notified about changes in a player's status. The handler is a block object that the voice chat object calls when a player connects or disconnects from the channel and when a player starts and stops speaking. For example, you might use the update handler to highlight that player's name in your user interface when he or she is speaking.

Listing 7-3 (page 72) shows an implementation of an update handler for player state.

Listing 7-3 Receiving updates about the player

```
teamChat.playerStateUpdateHandler = ^(NSString *playerID, GKVoiceChatPlayerState
state) {
     switch (state)
     {
         case GKVoiceChatPlayerSpeaking:
            // insert code to highlight the player's picture.
             break;
```

```
        case GKVoiceChatPlayerSilent:
           // insert code to dim the player's picture.
            break;
     }
  };
```

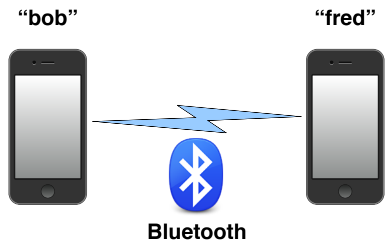
Peer-To-Peer Connectivity

This part of *Game Kit Programming Guide* describes how to use peer-to-peer connectivity in your application.

Peer-to-Peer Connectivity

The `GKSession` class allows your application to create and manage an ad-hoc Bluetooth or local wireless 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 iOS. Further, sessions offer all applications an exciting mechanism to allow users to collaborate with each other.

Figure 1 Bluetooth and local wireless 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 that have been discovered by the session or you can use a `GKPeerPickerController` object to present a standard user interface to configure a session between two devices.

Once a network between the devices is established, the `GKSession` class does not dictate a 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.

Requiring Peer-to-Peer Connectivity

If your application requires peer-to-peer connectivity, you should ensure that only users whose devices support it can purchase and download your application. To require peer-to-peer support, add the `peer-peer` key to the list of required device capabilities stored in your application's `Info.plist` file. See "iTunes Requirements" in *iOS App Programming Guide* .

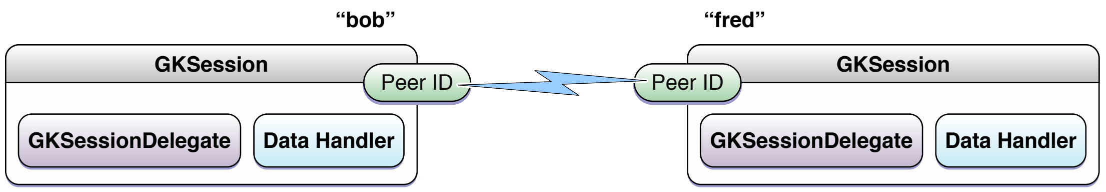
Sessions

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

Peers

iOS-based devices connected to an ad-hoc wireless network are known as **peers**. A peer is synonymous with a session object running inside your application. You need to be sure that each peer creates a unique peer identification string (peer ID), used to identify it to other peers on the network. Interactions with other peers on the network are done using peer IDs. For example, if one peer knows the peer ID of another peer, it can retrieve a user-readable name for that peer by calling the session object's `displayNameForPeer:` method, as shown in Figure 2.

Figure 2 Peer IDs are used to interact with other peers



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

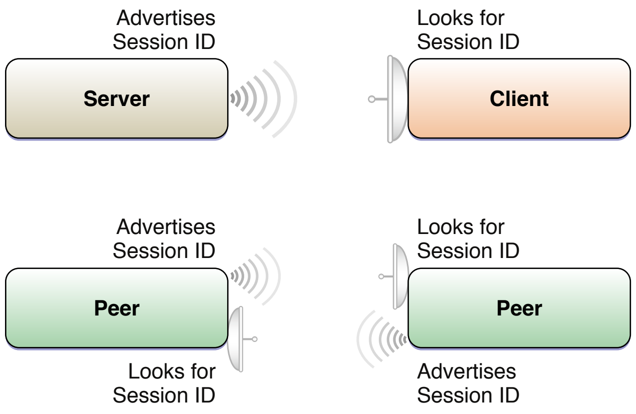
Discovering Other Peers

Every session implements its own specific type of service. This service 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 servers advertising themselves on the network; 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 that have a matching session ID.

Figure 3 Servers, clients, and peers



The session ID should be 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 an ID using the application's bundle identifier.

To establish a connection, at least one device must advertise as a server and another must search for it. Your application provides code for both advertising and searching. 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.

Note The maximum size of a client-server game is 16 players.

Implementing a Server

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

The servers is notified when a client requests a connection to the service. When a client sends a connection request, the `session:didReceiveConnectionRequestFromPeer:` method on the server's delegate is called. A typical behavior of the delegate should be to use the `peerID` string to retrieve a user-readable name by calling the `displayNameForPeer:` method. The server 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:` method or rejects it by calling the `denyConnectionFromPeer:` method.

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

An instance of your application acting as a client initializes the session by calling the `initWithSessionID:displayName:sessionMode:` method with a session mode parameter specifying either `GKSessionModeClient` or `GKSessionModePeer`. After configuring the session, your application searches the network for advertising servers by setting the session's `available` property to `YES`. If the session is configured with the `GKSessionModePeer` session mode it also advertises itself as a server, as described in "Implementing a Server" (page 77).

When a client discovers an available server, the client session's delegate receives a call to its `session:peer:didChangeState:` method; Game Kit provides the `peerID` string of the discovered server. Your application can call the session's `displayNameForPeer:` method 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 a 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 sent to other peers is 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). Messages larger than 1000 bytes may need to be split into smaller chunks and reassembled at the destination, incurring additional latency and overhead.

Note The largest message size allowed is 87 kilobytes. If you need to send more than that, you must split your data into multiple messages.

You can choose to send data **reliably**, whereby a session retransmits data any that has failed to reach its destination, or **unreliably**, whereby a session only attempts to send the data once. Unreliable messages are appropriate when the data must arrive in real time to be useful to other peers, and when sending an updated packet is more important than resending the data that failed to reach its intended recipient (for example, when sending dead-reckoning information).

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

In order for your application to receive data sent by other peers, you must implement 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 main thread of your application.

Important All data received from other peers should be treated as *untrusted* data. Your application should validate the data a peer receives from other peers; write your code carefully to avoid security vulnerabilities. See the *Secure Coding Guide* for more information.

Disconnecting Peers

To end a session, your application calls 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 nonresponsive for a period of time, that peer is automatically disconnected from the session. Your application can modify the `disconnectTimeout` property to control how long the session waits for a peer before disconnecting it.

The session delegate's `session:peer:didChangeState:` method is called when a peer disconnects from the session.

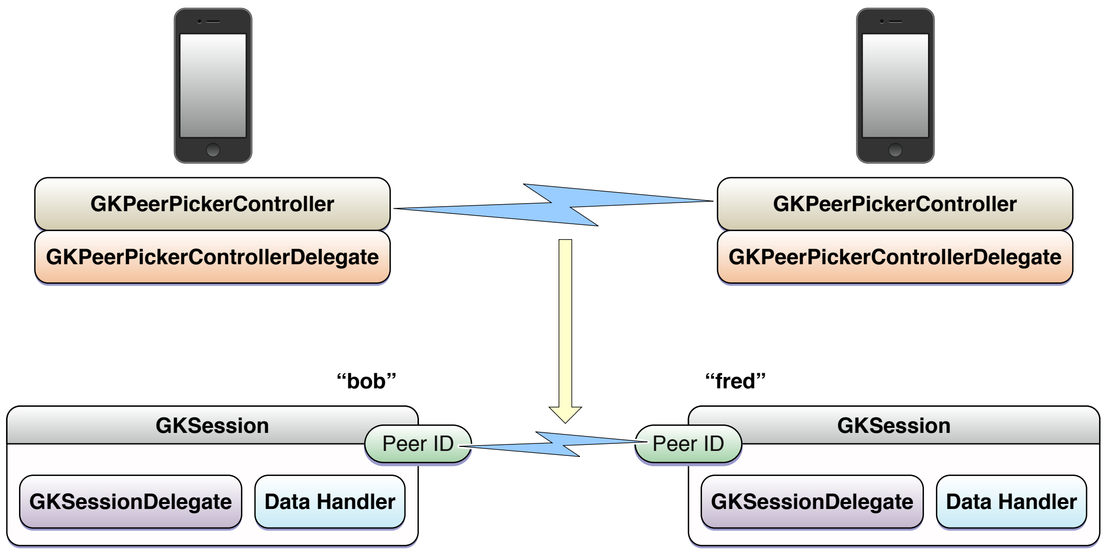
Cleaning Up

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

The Peer Picker

Although you may choose to implement your own user interface using the `GKSession`'s delegate, Game Kit offers a standard user interface for the discovery and connection process. A `GKPeerPickerController` object presents the a peer picker user interface that allows a user to create a peer-to-peer connection to another device. The `GKPeerPickerController` object returns 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 a 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 `connectionTypesMask` property is used to configure the list of available connection methods the application allows the user to choose from. A user 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 The peer picker creates only Bluetooth connections. If your application offers 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

Your application shows the peer picker 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 peer-picker 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 chapter, you should read "Working with Sessions" (page 85) to see what your application can do with the session created by the peer picker.

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 a 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. You can also also add Internet (online) connections to the connection types mask. If you add Internet connections, your application must also 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.
    }
}
```

4. If you want to provide your own custom session object (rather than the default object created by the peer picker controller), 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;
}
```

5. 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.
}
```

6. 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];
}
```

7. Add code to display the picker dialog in your application.

```
[picker show];
```

Working with Sessions

This chapter 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 82).

A peer's 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, make sure that your code follows the steps found in "Finding Peers with Peer Picker" (page 82), then continue with these steps.

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 state changes are handled automatically by the peer picker, but your 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 choose either to process the data immediately or retain the data and process it later within your application. Your application should not perform lengthy computations within this method.

4. When it is time to end the connection, disconnect from the other peers and release the session object.

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

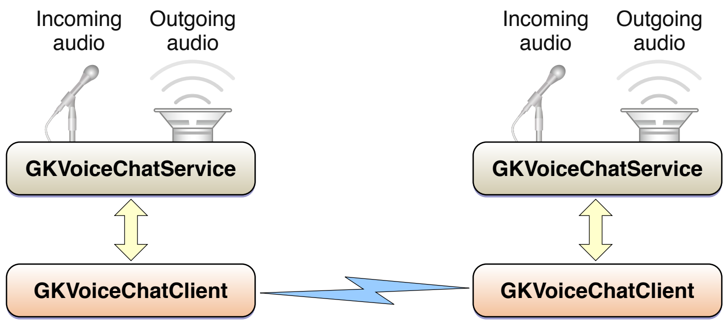
In-Game Voice

This part of *Game Kit Programming Guide* describes how to add In-Game Voice support to your application.

In-Game Voice

A `GKVoiceChatService` object allows your application to easily create a voice chat between two iOS-based devices, as shown in Figure 1. The voice chat service listens to the microphone, sends data it records to the other participant, and uses the device's audio subsystem to play audio data 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

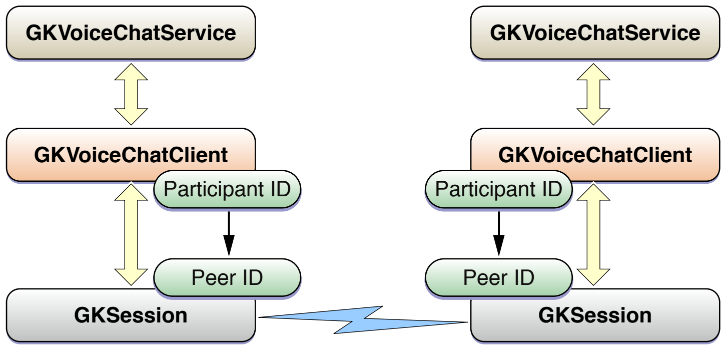
Each participant in a voice chat is identified by a unique **participant identifier** string provided by your client. The format and meaning of a participant identifier string is left to your client to decide.

Discovering Other Participants

The voice chat service uses the client's network connection to exchange configuration data between the participants in order to create a direct connection between the two devices. 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.

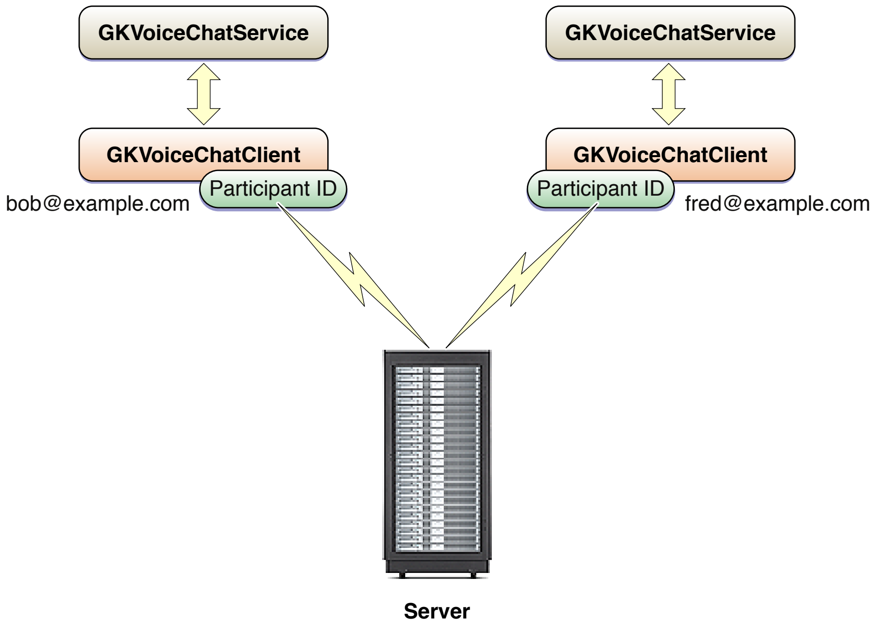
For example, if your application on one device is already connected to your application on another device through a `GKSession` object (see "Peer-to-Peer Connectivity" (page 75)), then each peer on the network is already uniquely identified by a `peerID` string. The session already knows the `peerID` 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 by their email addresses and can route data between them.

Figure 3 Server-based discovery



Depending on the design of the server, the server can provide a list of participant identifiers to the clients or or may require the client to provide the participant identifier (email address) of another user. In either case, the server is used as an intermediary to transmit data between the two users.

When the voice chat service on one device wants to send its configuration data to a participant on another device, 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 the data to the service by calling the service's `receivedData:fromParticipantID:` method. The voice chat service uses this exchange of data to configure its own real-time network connection between the two participants. The voice chat service uses the previously existing connection only to transfer enough data to create its own chat 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 initiates the chat by calling the voice chat service's `startVoiceChatWithParticipantID:error:` method with the `participantID` of another participant. The service uses the client's network as described in "Discovering Other Participants" (page 88)to request a new chat.

When a voice chat service on a device receives a connection request, it calls the client's `voiceChatService:didReceiveInvitationFromParticipantID:callID:` method to handle the request. The client accepts the chat request by calling the service's `acceptCallID:error:` method, or rejects it by calling `denyCallID:`. You might want your application to prompt users first to see whether 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 (on either device) to end a voice chat. You should make sure your application also stops the chat if it discovers that the other user is no longer available.

Controlling the Chat

Once the two participants are connected in a voice chat, speech is automatically transmitted between the two iOS-based devices. 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 monitor the volume level of the user. Similarly, your application can monitor the volume level received by the other user by setting `outputMeteringEnabled` to `YES` and reading the `outputMeterLevel` property. To improve application performance, your application should enable metering only when it expects to read the meter levels of the two participants.

Adding Voice Chat

In order to implement voice chat, your application must be running on two different devices that have a network connection between them. This example walks through one possible implementation of voice chat, using a `GKSession` object to create a network to the other device. For more information on `GKSession` objects, see "Peer-to-Peer Connectivity" (page 75).

To implement this example, 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. Because a session's `peerID` string already uniquely identifies the peer, the voice chat 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 voice chat service calls the client when it needs to send data to other participants in the chat. Most commonly, the service calls the client to establish its own real-time connection with other participants. As both the `GKSession` and `GKVoiceChatService` classes use an `NSData` object to hold message data, you simply pass the `NSData` object received from the voice chat service to the session.

If the connection is used for both voice chat and to transmit your own information, you need to provide additional information in the message you transmit via the session so that you can differentiate your data transmissions from those sent by the voice chat service.

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];

}
```

The receive handler 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 after the session's connection is established, or you might offer the user the option of creating a voice chat. 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 a 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
2011-03-08	Revised the process for authenticating the local player. Clarified many aspects of Game Center usage.
2010-10-25	Clarified the requirements for performing server-based matchmaking. Improved the guidelines implementing voice chat in a Game Center application.
2010-08-27	Updated to add Game Center classes.
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.