Game Kit Programming Guide

Networking & Internet

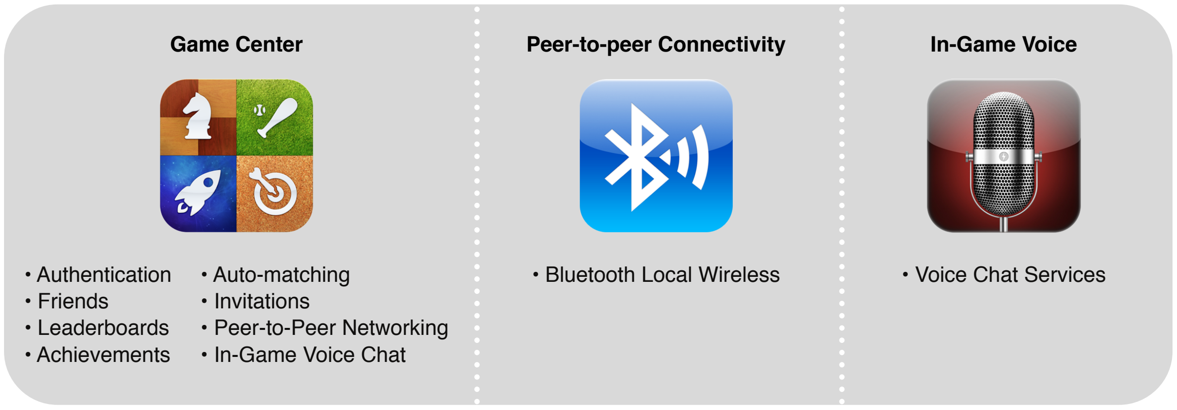
2010-10-25

Contents

Figures, Tables, and Listings

About Game Kit

Game Kit offers features that you can use 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 encourages more players to download and play those games. Positive word-of-mouth provides the best kind of free advertising to your game — happy customers.



At a Glance

Game Kit consists of three different technologies, each of which can be adopted independently.

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 games. Game Center provides its services over both wireless and cellular networks. These services include:

- Players connect to Game Center and interact with other players by creating an **alias**. Players can set status messages as well as mark other players as friends.
- **Multiplayer** allows your application 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.
- **Leaderboards** allow your application to record and fetch player scores from Game Center.
- **Achievements** track a player's accomplishments in your game. Achievements are tracked by Game Center and can be viewed in the Game Center application as well as inside your application.

Game Center is supported in iOS 4.1 and later.

Relevant Chapters: "Game Center Overview" (page 15), "Working with Players in Game Center" (page 21), "Leaderboards" (page 27), "Achievements" (page 33), "Multiplayer" (page 41), "Using Matches to Implement Your Network Game" (page 53), and "Adding Voice Chat to a Match" (page 59)

Peer-to-peer Provides Local Wireless and Bluetooth Networking

Peer-to-peer connectivity allows your application to create an ad-hoc Bluetooth or local wireless network between multiple iOS 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: "Peer-to-Peer Connectivity" (page 65), "Finding Peers with Peer Picker" (page 71) and "Working with Sessions" (page 73)

In-game Voice Provides a Common Voice Chat Infrastructure

In-game voice allows your application to provide voice communication between two iOS devices. In-game Voice samples the microphone and handles mixing of audio playback for you. In-game voice relies on your application to provide a network connection. It 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: "In-Game Voice" (page 77) and "Adding Voice Chat" (page 81)

How to Use This Document

If you want to add Game Center support to your application, first read "Game Center Overview" (page 15). Then read "Working with Players in Game Center" (page 21) to learn how to authenticate a player on the device. Once your application authenticates players, read the other chapters in Part I for the details on adding leaderboards, achievements or matchmaking to your application.

If you want to add local peer-to-peer networking to your application, read "Peer-to-Peer Connectivity" (page 65) 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 77) for an overview of the `GKVoiceChatService` class, and "Adding Voice Chat" (page 81) 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 Center-aware application, 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.

Game Center

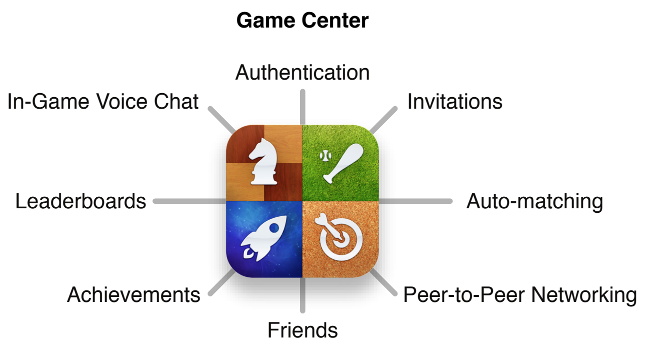
This part of *Game Kit Programming Guide* describes how to use Game Center in your application.

PART I

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 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 application is a new built-in application 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 application and the Game Center application 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 application.

So what features does Game Center provide?

- Players create and manage an account that provides them an online persona, also known as an **alias**. A player uses an account 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 application to post scores to the Game Center service where they can later be viewed by players. Scores are viewed by launching the Game Center application; your application 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 different 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 application 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 application; they can also compare their achievements with those earned by a friend. Your application 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 application can have either use Game Kit to connect all the participants together, or have Game Kit deliver a list of players to you. In the latter case, you would provide your own network implementation that connects the players to a server you provide.

  Matchmaking provides both automated matching and invitations. With automated matching, your application creates a request that specifies how many players should be invited into the match, and which 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 application or from within the Game Center application. When a friend is invited, they receive a push notification that allows them to launch your game and join the match. A push notification can be sent to a player even if they do 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 application 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 purchase your application immediately. By connecting game players to each other, word-of-mouth can quickly spread news about your game.

Essential Game Center Concepts

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

All Game Center Applications 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 application reports scores to a leaderboard, they are always reported for the authenticated player. Most Game Center classes only function if there is an authenticated player. Before your application uses any Game Center features, it must authenticate the local player. Authenticating the local player 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 application 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 application attempts to authenticate the local player. Your application must handle any authentication errors by disabling all Game Center features.

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

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

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 application, see *View Controller Programming Guide for iOS*.

Most Game-Center Classes 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 application posts a request by calling Game Kit. Later, when the operation completes (or fails because of an error), Game Kit calls your application 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 application requests, the block also receives other parameters to allow Game Kit to return information to your game. You should be comfortable with blocks before attempting to add Game Center to your application.

Sending Data to Game Center Must Be Retried After a Network Failure

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

Receiving Data from Game Center May Receive Cached Data After a Network Failure

When your application retrieves data from Game Center (for example, 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, it is possible to receive both an error and partial results from Game Kit.

Checklist for Implementing a Game Center-aware Application

When you are ready to add Game Center support to your application follow this checklist:

- Enable Game Center on iTunes Connect. See "Configure the Application in iTunes Connect" (page 18).
- Configure the Bundle Identifier for your application. See "Configure Your Application's Bundle Identifier" (page 18).
- Link to the Game Kit framework.
- Import the `GameKit/GameKit.h` header.
- If your game requires Game Center, add the Game Center key to the list of capabilities required by the device. See "Requiring Game Center in Your Application" (page 19).
- If your game does not require Game Center, you should weak link to the Game Kit framework. Then, when your application launches, test to make sure that Game Center is supported. See "Optionally Using Game Center In Your Application" (page 19)
- 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 21).

Configure the Application in iTunes Connect

Before you add any Game Center code to your application, you need to set up your application on 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.

Configure Your Application's Bundle Identifier

A **bundle identifier** is a string you create for your application (such as `com.myCompany.myCoolGame`). The bundle identifier is used throughout the application development process to uniquely identify your application. For example, when you manage your application on iTunes Connect, the bundle identifier is used to match your XCode application to the profile you create there.

Normally, you set the bundle identifier in your Xcode project prior to shipping your application to customers; you can start development without setting the bundle identifier. This is *not* the case when you develop a Game Center application. Game Center uses the application's bundle identifier to retrieve the application's data on 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 Application

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 do this, add the `gamekit` key to the list of required device capabilities in your application's `Info.plist` file. See "iTunes Requirements" in *iOS Application Programming Guide*.

Optionally Using Game Center In Your Application

If your application wants to use Game Center on supported devices, but does not require Game Center to function properly, you can weak link to the Game Kit framework and test for it at runtime.

Listing 1-1 (page 19) provides code you should use to test for Game Center support. This code 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.

Listing 1-1 Testing for Game Center

```
BOOL isGameCenterAvailable()
{
    // Check for presence of GKLocalPlayer API.
    Class gcClass = (NSClassFromString(@"GKLocalPlayer"));

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

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

Note: Game Center was first previewed in iOS 4, 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 iOS 4 devices.

Testing a Game Center Application

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

As a developer, your Game Center account may be logged into either Sandbox for testing, or into the live environment but not both. To change which account you are logged into, switch to the built-in Game Center app and log out, then run any Game Center enabled app. If that app is provisioned for development, enter your test account information to log into Sandbox. Otherwise, enter your live account information to log 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 reuse an existing Apple ID.

Table 1-1 Determining which environment your application is executing in

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 share 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 build those features into your game, you need to understand how Game Center works with players.

Players Must Create Accounts to Access Game Center

A player that wants to take advantage of Game Center support in their games creates a Game Center account. A Game Center account serves many roles: it allows the user 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. For example:

- A high score in a leaderboard is always associated with a player.
- Whenever a player makes progress on an achievement, you store that information on Game Center.
- A player can list other players as friends. Your application can retrieve information about a player's friends.

Game Center assigns a string to each account, known as a **player identifier**, that uniquely identifies that specific account. Although a user can change other information, their player identifier never changes. Player identifiers are used throughout Game Kit whenever a player needs to be identified. For example, when your application sends a message to another player connected in a match, the message is addressed using that player's identifier.

Your application should use player identifiers when you need to associate information with a specific player. For example, if you store information on the device to track a player's saved progress in your game, you can use player identifiers to disambiguate between multiple users. This allows multiple users to use the same device while allowing each to have their own saved games. Similarly, if your application connects to your own network services, you can use the player identifier on your service to save data there as well.

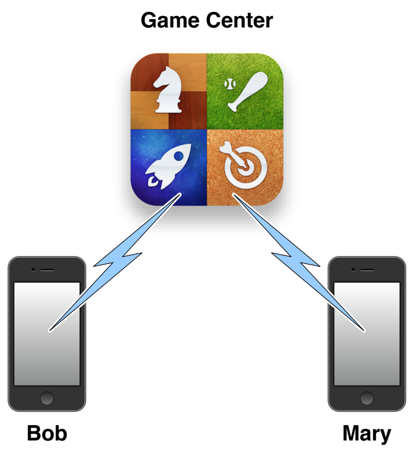
Important: Do not make assumptions about the contents of the player identifier string. Its format and length are subject to change.

One thing that you should never do is display player identifiers to the user; they are not intended to be shown 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**. Given a player identifier, you can retrieve a player's alias from Game Center and display that to the player.

The Local Player is the Player Signed into the Device

Game Kit can provide you with information for any player on the service, but one player is always essential to your application. 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



Most Game Center classes implicitly reference the local player. For example, when your application reports scores to a leaderboard, it always reports scores earned by the local player. Your application should never work with Game Center classes when there is not an authenticated local player.

Authenticating the Local Player

When a Game Center-aware application launches, it should authenticate the local player as soon as possible after launching; ideally this should be done as soon as your application can present a user interface to the player. If a player has already authenticated their account (for example, they may have authenticated in another game or by using the Game Center application), control returns immediately to your application. If there is no authenticated player, Game Kit displays an authentication screen, allowing the player to sign in with an existing account or create a new account. Authenticating the local player transparently handles creating new Game Center accounts.

"Retrieving Information About Remote Players" shows how to authenticate the local player. This method could be called in your application delegate's `didFinishLaunchingWithOptions:` method, or shortly after that. To authenticate the player, the method retrieves the shared instance of the `GKLocalPlayer` class and calls its `authenticateWithCompletionHandler:` method. The parameter is a block object to be called when the authentication process completes.

Listing 2-1 Authenticating the local player

```
- (void) authenticateLocalPlayer
{
    [[GKLocalPlayer localPlayer] authenticateWithCompletionHandler:^(NSError
*error) {
         if (error == nil)
         {
              // Insert code here to handle a successful authentication.
         }
         else
         {
             // Your application can process the error parameter to report the
 error to the player.
         }
     }];
}
```

If an error occurred, then there is no authenticated player. Game Kit returns an error to your block that describes the error. When an error occurs, present it to the user and disable any other use of Game Center in your application.

If the error parameter is `nil`, then authentication completed successfully. Your application can now use other properties on the local player object and configure other Game Center-related classes. For example, your application might perform any or all of the following steps in the completion handler:

- Read the local player object's `alias` property to retrieve the local player's alias.
- Retrieving a list of player identifiers for the local player's friends. See "Retrieving Identifiers For the Local Player's Friends" (page 25)
- Retrieve the local player's previous progress on achievements. See Listing 4-2 (page 36).
- Add an invitation handler to receive matchmaking requests. If your application supports matchmaking, you must perform this step immediately after authenticating the player; your application may have been launched specifically to handle an invitation. See "Processing Invitations from Other Players" (page 44)

Register a Notification to Detect if the Local Player Disconnects From Game Center

In iOS 4 and later, multitasking allows multiple applications to run on the device simultaneously. When your game moves into the background, the player may log out of their Game Center account. Another player may have logged in. The local player may not stay authenticated even while your application is running. For this reason, all Game Center applications must register a notification handler so that they can be informed when the local player disconnects from Game Center.

The `GKLocalPlayer` class defines the `authenticated` property that declares whether there is currently an authenticated local player. The `GKPlayerAuthenticationDidChangeNotificationName` notification is posted whenever the `authenticated` property changes.

"Retrieving Identifiers For the Local Player's Friends" shows how to register for the notification that Game Center posts when the player's authentication state changes.

Listing 2-2 Registering for the `GKPlayerAuthenticationDidChangeNotificationName` notification

```
- (void) registerForAuthenticationNotification
{
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc addObserver: self
           selector:@selector(authenticationChanged)
               name:GKPlayerAuthenticationDidChangeNotificationName
             object:nil];
}
```

Listing 2-5 provides a structure for your notification handler. This handler tests the `authenticated` property and updates the application's internal state based on the change in value. The actual work to perform here is dependent on your application. Usually, your application should remove invitation handlers and dispose of other objects that rely on the local player.

Listing 2-3 Implementing the notification handler

```
- (void) authenticationChanged
{
    if ([GKLocalPlayer localPlayer].isAuthenticated)
        // Insert code here to handle a successful authentication.
    else
        // Insert code here to clean up any outstanding Game Center-related
classes.
}
```

Retrieving Details About Players

You are already familiar with the `GKLocalPlayer` object, representing the currently logged in user. However, `GKLocalPlayer` is actually a subclass of `GKPlayer`, which is used in Game Kit to describe any player on Game Center. A player object's properties include the user's alias, so in most cases your application needs player objects to populates its user interface. This section describes how your application creates player objects.

Retrieving Information About the Local Player

As long as the local player is authenticated, the other properties on the local player object are loaded with appropriate values for the local player, with the exception of the `friends` property, which must be explicitly loaded from Game Center. Your application can access these `GKPlayer` properties:

- The `playerID` property holds the player's player identifier string.
- The `alias` property holds the user-visible string chosen by this player.

The `GKLocalPlayer` object adds additional properties:

- The `friends` property is populated with the identifier strings of the local player's declared friends. This property is not automatically filled after authentication. See "Retrieving Identifiers For the Local Player's Friends" (page 25).
- The `underage` property states whether this user is underaged. Some Game Center features are disabled when the value of this property is `YES`. This property is provided so that if your application also wants to disable its own features, it may do so.

Retrieving Identifiers For the Local Player's Friends

Game Center is fundamentally intended to be a social experience. Game Center allows a player to mark other players as friends, invite friends into matches, and otherwise communicate with friends. Game Kit allows your application to access information about the local player's friends.

Retrieving details about the local player's friends is a two-step process. First, your application calls Game Kit to retrieve an array of player identifiers for the local player's friends. Then your application calls Game Kit with this list of identifiers to receive a player object for each.

Listing 2-4 provides a method that retrieves the player identifiers for the local player's friends. This code retrieves the local player, ensures that the local player is already authenticated, and calls the `loadFriendsWithCompletionHandler:` method. After the identifier strings are retrieved from Game Center, the local player object sets its `friends` property to include this array of identifiers and returns the array to your application's completion handler.

Listing 2-4 Retrieving the local player's friends

```
- (void) retrieveFriends
{
    GKLocalPlayer *lp = [GKLocalPlayer localPlayer];
    if (lp.authenticated)
    {
       [lp loadFriendsWithCompletionHandler:^(NSArray *friends, NSError *error)
 {
             if (error == nil)
             {
                   // use the player identifiers to create player objects.
             }
             else
             {
                   // report an error to the user.
             }
         }];
    }
}
```

Retrieving Information About Remote Players

Whether you received player identifiers by loading the identifiers for the local player's friends, or from another Game Center class, you must retrieve the details about that player from Game Center. Listing 2-5 (page 25) shows how to pass an array of player identifiers to Game Kit to retrieve player objects..

Listing 2-5 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.
          }
      }];
}
```

When your completion handler is called, you can retrieve the common properties for each `GKPlayer` object.

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 used to record scores earned by anyone who plays your game. Your application 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 application. 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 application, you need to do the following:

- Before you can add leaderboards, first add code to your application to authenticate the local player. See "Working with Players in Game Center" (page 21).
- Decide how your game calculates scores. You are free to design your own scoring mechanisms that are appropriate to your game.
- Go to iTunes Connect and configure the leaderboards for your application. In this step, you decide how the scores that your application posts to Game Center are formatted when they are displayed in a leaderboard. Leaderboards can be localized for different languages and regions. See "Configuring a Leaderboard on iTunes Connect" (page 27).
- Add code to report scores to Game Center. See "Posting Scores to Game Center" (page 29).
- If your application cannot report a score to Game Center, it 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 29).
- While a player can view the high scores in the Game Center application, you should allow the 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 30). 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 31).

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 the leaderboard to your application, you configure leaderboards on iTunes Connect to tell Game Center how a score 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 you having to write any code.

Each application has its own leaderboard configuration; leaderboard configurations and score data are never shared between multiple applications.

Configuring a leaderboard requires you to make some decisions:

- What units are your scores measured in?
- Are score values ranked in ascending or descending order?
- How are scores formatted?
- Are all scores reported by your application included in a single list, or do different modes of play in your game 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 leaderboard 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, a time value, or 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 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". Another game 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 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.

Leaderboard Categories

Games often define more than one mode of play. For example, a mode might specify how difficult the game is(easy, medium, hard), the game's victory conditions (Capture the Flag, Last Man Standing), or even specific maps the game provides. When configuring your game, you may want distinct game modes to be have their own leaderboards. To do this, you use a **category**, which is simply an abstraction used to separate the scores into different groups. Your application creates one or more categories. You decide what they actually mean to your application. Each game may create up to 25 categories.

To define a category, you create a **category identifier** string and provide localizations for the category's title. A category identifier is a string that uniquely identifies the category. You create category identifiers on iTunes Connect, and your application uses the same category identifiers when it reports scores to Game Center. For example, you might create a category of "mygame.easy" as your category ID and "Easy Mode" as the title to be displayed in the leaderboard.

If your game supports more than one category, you can choose whether Game Center shows an aggregate leaderboard; if you allow this, the aggregate leaderboard shows scores regardless of which category they were reported to.

Important: All categories share the same score format.

Posting 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 category for the leaderboard it should be reported to, then the code sets the value property to the score the player earned. The category must be one of the category identifiers you defined when you configured your leaderboards in iTunes Connect. This code does not set the player who earned the score or the time the score was earned; those properties are automatically set when the score object is 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 de-archived when it launches.

Showing the Standard Leaderboard

In addition to reporting scores, your application should also allow players to see their scores. The simplest way to do this is with a `GKLeaderboardViewController` object. A leaderboard view controller provides a similar user interface to the leaderboard found in the Game Center application. The leaderboard view controller is presented modally by a view controller implemented 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 Showing the default leaderboard

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

Before presenting the leaderboard, your application can take additional steps to configure the leaderboard view controller:

- 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 30) shows how your view controller dismisses the leaderboard.

Listing 3-3 Responding when the user dismisses the leaderboard

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

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

Retrieving Leaderboard Scores

If your application wants to examine the score data or create a custom leaderboard view, your application can directly load score data from Game Center. To do this, 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 initializes a new leaderboard object and configures the `playerScope`, `timeScope`, and `range` properties to grab the top ten scores from all players, 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 associated with a match to load their 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 Information About Remote Players" (page 25). The value property holds the actual value you reported to Game Center. More importantly, the formattedValue property provides a string with the score 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 on iTunes Connect. Listing 3-6 (page 32) 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 using a block.

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 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 both in the Game Center application and inside your application.

Checklist for Supporting Achievements

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

- Before you can add achievements, you should first add code to your application to authenticate the local player. See "Working with Players in Game Center" (page 21).
- 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 33).
- 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 34).
- 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 36).
- While a player can view achievements using the Game Center application, you should allow the player to view achievements from within your application. See "Displaying Achievements Using an Achievement View Controller" (page 38) 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 38) 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 the player accomplishes that goal within your game. What those goals are and how those goals are accomplished vary for different genres. 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 and some achievements should only be earned with significant skill and effort. This encourages players to improve their skills. When they earn difficult achievements, your game acknowledges their newly earned level of skill.

When you are ready to create the achievement, you need define the achievement on 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, and your application uses the same string when it wants to report progress on an achievement. Identifiers are never shown to the player.
- A **title** string that names the achievement.
- Two **description** strings for the achievement. Regardless of how you set an achievement's goals and difficulty level, your description 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 In App Purchase, you may want to reserve some of your point budget for achievements that apply to the In App Purchase content.
- An **image** to be displayed when a player complete's the achievement. You only create an image for the completed achievement. Incomplete achievements always display a standard image provided by Game Center.
- Finally, decide whether the achievement is visible to the user when they first launch your game or it if 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 only available when the player purchases additional content.

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

All the data for your achievements is edited using your iTunes Connect account. Descriptions and titles are also localized for whichever 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 on Game Center, you allow the Game Center application 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 reached 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, 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 retain the achievement object (possibly adding it to an array). Then, 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

The previous example created a new achievement object, but often your application needs to know whether the user has already made progress on an achievement. Your application can retrieve the player's progress 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 place to load the player's progress is immediately after the player is authenticated. Listing 4-2 (page 36) 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.
        }
       }];
}
```

In practice, your application may either need to create a new achievement object or use one initialized with data fetched from Game Center. If an achievement has been reported before, you should use the object loaded from Game Center. Otherwise, your application should create a new achievement object. A mutable dictionary is an ideal way to store a collection achievements objects, adding new objects as necessary. The `identifier` property is useful as the dictionary key. Here's how to modify the previous example:

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

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

2. Initialize the achievements dictionary.

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

3. When your game loads achievement data, 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 an achievement object, creating a new one if one is not found:

```
- (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];
}
```

From now on, you never create an achievement object directly; you call `getAchievementForIdentifier:` instead.

Resetting Achievement Progress

Your application may want to allow the player to reset their progress on achievements. To do this, you call the `resetAchievementsWithCompletionHandler:` class method. Listing 4-3 (page 37) demonstrates how to do this. First, it clears any locally cached achievement objects created by the previous example. Next, 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 now hidden again. This means, for example, if your application want to report progress on certain achievements to show them again. For example, if the only reason the achievement was hidden was because it was associated with In App Purchase content, 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, it must be presented modally by another view controller.

Listing 4-4 (page 38) shows how your view controller can present an achievements screen. It creates a new achievements 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 implement the `GKAchievementViewControllerDelegate` protocol. The delegate is notified when the user dismisses the achievements view. 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 customize the appearance of your achievements screen, you can the achievement descriptions and combine them with the progress information you previously loaded to present your own custom user interface. Game Kit provides achievement descriptions to your game using the `GKAchievementDescription` class. A `GKAchievementDescription` object's properties correspond to the information you added on 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 completed achievement, your application loads the completed achievement image. This allows your application to only load images you need, which reduces your memory footprint.

Listing 4-6 shows how to load the achievement descriptions. It does this by 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 `image` property of achievement description object 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 common 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 provided by Game Center. Matchmaking makes it easy for like-minded players playing your game to discover each other and connected into a network.

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:

- Before you can add matchmaking, you must first add code to your application to authenticate the local player. See "Working with Players in Game Center" (page 21).
- Add code to display the matchmaking screen to the user. You do this by creating a match request and using it to initialize a matchmaker view controller. See "Creating a New Match Starts With a Match Request" (page 42) to learn about match requests and "Presenting the Matchmaking User Interface" (page 43) to learn how to display the view controller..
- Add code to process invitations. Game Kit calls your invitation handler whenever it has a pending invitation for the local player. See "Processing Invitations from Other Players" (page 44).
- Optionally, add code to programmatically find matches. See "Finding a Match Programmatically" (page 45).
- Read "Advanced Matchmaking Topics" (page 46) for a list of advanced matchmaking functionality you might want to add to your game. For example, if your game offers multiple game modes (such as Capture the Flag or Last Mand Standing), you can restrict matchmaking so that players only find other 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 and already has your application installed, their device automatically launches your application (or moves it to the foreground if it was already running). Once all the friends are in the match, the players start the match. On each device, your game is notified that a match is ready. Each copy of the 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 application. 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!
- In both of the previous scenarios, 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 programmatic auto matching to create a match. When you use programmatic auto 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 other players waiting to join a match. Programmatic auto matching does not show any user interface; you can customize what the player sees while the match is being formed.

Note: Matchmaking can only be done 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 Kit uses the match request to customize its behavior. For example, when your application displays the standard matchmaking interface, Game Kit uses the request to customize the screen's appearance. If, on the other hand, your application uses the programmatic approach, the match request is used to find compatible players to gather into a new match.

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

Listing 5-1 Creating a match request

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

Note: In iOS 4.1, `minPlayers` must be at least `2` and `maxPlayers` must be equal 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 50) 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 auto matching.

Listing 5-2 provides code that 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 presents it to the player. After the view is presented, 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. The player canceled before a match could be created, so most games would simply return to a previous user interface screen.

Listing 5-3 Implementing the `matchmakerViewControllerWasCancelled:` 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. For example, if the device 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 `matchmakerViewController:didFailWithError:` method

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

Finally, when the match was created and everyone is ready to start, your delegate receives a call to let you know the match was successfully created. The view controller returns a `GKMatch` object to your application. Listing 5-5 (page 44) assigns the match object to a retaining property, and then uses it to start the match.

The `GKMatch` class is covered in detail in "Using Matches to Implement Your Network Game" (page 53).

Listing 5-5 Implementing the `matchmakerViewController:didFindMatch:` method

```
- (void)matchmakerViewController:(GKMatchmakerViewController *)viewController
didFindMatch:(GKMatch *)match
{
    [self dismissModalViewControllerAnimated:YES];
    self.myMatch = match; // Use a retaining property to retain the match.
    // Start the game using the match.
}
```

Processing Invitations from Other Players

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

An invitation handler performs activities similar to hosting a match— in fact, you can probably 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 the application receives an invitation directly from another player. In this situation, the other player's application has already created the match request, so this copy of your application does not need to create one.
- The `playersToInvite` parameter is non-`nil` when your application is launched directly from the Game Center application to host a match. This parameter holds an array of player identifiers for the players to invite to the game. Because this instance of the application is hosting the match, it must create a new match request. It assigns the match request's `playersToInvite` property to the list of players passed into the invitation handler. When the matchmaking screen is displayed, it is pre-populated with the list of players already included in the match.

Important: Your application should set the invitation handler as early as possible after your application is launched; an appropriate place to set the handler is in the completion block you provided 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. It cleans up any matches already in progress, instantiates a new view controller object based on whichever input parameter was non-`nil`, 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 find a match for the player without displaying the standard user interface. For example, your application could offer an "instant match" button to let someone join a new match by pressing a single button.

Listing 5-7 shows how to find a match programmatically. As in the earlier examples, the code first creates a match request. It then retrieves the matchmaker singleton object and asks to find a match. The completion handler called when a match is found or if an error occurs. If a match was returned, then the block assigns the match object to a retaining property and uses it 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.
            // Start 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 an replacement, instead of aborting the match in progress.

To do this, you use code similar to that found in Listing 5-7 (page 45), but instead of calling the `findMatchForRequest:withCompletionHandler:`, your application calls the `addPlayersToMatch:matchRequest:completionHandler` method, adding the additional parameter for the match to add the players to.

Canceling a Match Request

If you add the ability to programmatically find matches to your application, you should present your own user interface to the player that includes a way for them to cancel the match request. Listing 5-8 shows how your game terminate a pending match 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 following advanced features into your application:

- **Player groups** allow you to create subsets of players. Only players in the same group are auto matched with each other. See "Player Groups" (page 47).
- **Player attributes** allow your game to define specific roles that a player can fill in your game. When a match request is configured with a player role, auto matching only places them in a game that requires a player that fills that role. See "Player Attributes" (page 48).

- **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 hint to the player about whether they will quickly find a match. See "Searching for Player Activity" (page 49).
- **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 41)

Each of these advanced options can be used individually, or combined in the same application.

Player Groups

Game Center's default behavior is to auto-match any player waiting to play your game. 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. For example, you might want to group players into smaller groups. For example:

- Separate players by player skill level.
- Separate players by the rules set used to adjudicate the match (Capture the Flag, Last Man Standing, etc.).
- Separate players based on the map the match is played on.

You implement this in your application using player groups. With player groups, 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. Your application can place whatever value it wants in this property; Game Center does not care how you decide the value that goes there. However, when you provide a non-zero value, the player is only added to a match with players sharing the same 32-bit value.

Listing 5-9 initializes a match request and adds a player group. In this example, the player group value is calculated as a mask by performing an OR operation between a constant for the map the game is played on and a constant for the game rules.

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

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

Typically, your game should present its own user interface to allow the player to select parameters that are used to calculate the player group number.

Player Attributes

Some games offer different roles that a player can play in the game. For example:

- Many roleplaying games offer classes. Each class represents a different set of abilities that player brings to the match.
- In a sports game, a role might be a position on the field (quarterback, goalie, etc.).

Using player attributes, you can define a match request with the role that player wants to fill in the match. When Game Center automatches the player with other players waiting to start a match, it only matches the player into a match that needs a player to fill that role.

Player attributes have some limitations:

- Only one player may fill each role.
- All roles must be filled for the match to start.
- Each player may only ask to fill a single role.
- Roles are only checked 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 matching user interface. 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 that 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 play in your game. It is up to you to add a custom user interface in your game that allows the player to select the role they want to play. You set the `playerAttributes` property on the mask request to match the player's selection, and use this request to initialize a view controller or perform auto matching using the `GKMatchMaker` class.

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 auto 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 a match request that has a non-zero player attributes value. It adds a player to the match only if the requesting player's attributes mask does not overlap with the match's current mask. That is, if the two values are logically ANDed together, the result must be `00000000h`.

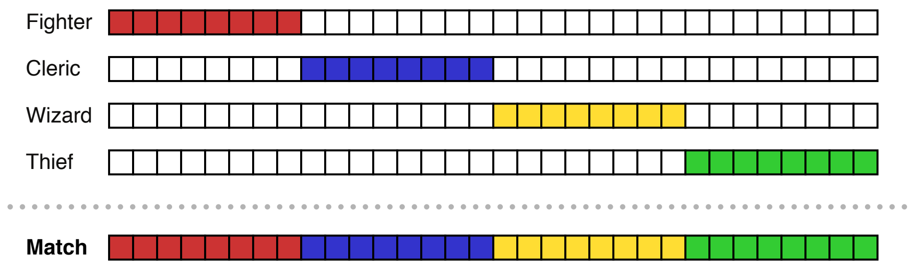
3. After adding a player to the match, the value of that player's player attributes value is logically ORed into the match's mask.

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 roleplaying game with four classes: Fighter, Wizard, Thief, and Cleric. Every group must have one and only one of each. To implement this, your application creates masks for each class that match the algorithm above. Listing 5-11 (page 49) 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 masks overlap; if any two masks are logically ANDed together, the resulting value is always `00000000h`. When all four masks are logically ORed 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 when matchmaking may take longer to complete. 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. This 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, Game Kit creates a match — an instance of the `GKMatch` object on each device — to connect all of the devices together. For most games, this is exactly what you want. The `GKMatch` class abstracts the network and does all the work to transmit data and voice between participants of the match. However, some games either need to support more players or need their own server to arbitrate the match. In this case, you can use matchmaking to find players for a **hosted match**. A hosted match does not create a `GKMatch` object. Instead, your application receives the player identifiers for all of the players in the match.

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

- You must design and implement your own networking code to connect each device to your server.
- Your application must design and implement its own networking protocol to inform other devices of the state of any participant in the match.
- If your application uses the standard matchmaking user interface, each device must inform Game Kit when a player connects to your server. This allows Game Kit to update the user interface.
- When participants need to send data to each other, your server uses its established connections to route the packets. Your implementation typically does this by converting a player identifier to the connection that player's device established to the server.
- Voice chat is not provided. However, the `GKVoiceChatService` class can be used to implement voice chat using the network connection you provided. See "In-Game Voice" (page 77).

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 receive a hosted match.

Whether your application creates the match by displaying the standard matchmaking user interface or creates a match programmatically, the behavior is similar. Instead of getting back an initialized `GKMatch` object, your application instead receives a list of player identifiers for the players to be connected into the match. Your game client should 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 screen to the player. Listing 5-13 modifies the method provided in Listing 5-2 (page 43)

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];
}
```

In addition to presenting the user interface, each device must also connect to your server. When the device connects to the server, your server should inform all connected participants. Each connected participant 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 is called to start the game. Previously, 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 peer-to-peer matches. You create a match request, get the shared matchmaker singleton, and use it to find the match. To create a hosted match, you call the matchmaker's `findPlayersForHostedMatchRequest:withCompletionHandler:` method instead. This method 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 Kit 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:

- First, you should have already written code to use the matchmaking service to create a match. The matchmaking services establish 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 41) for more information on finding matches.
- Design the network messages your application uses to communicate with other participants in the match. See "Designing Your Network Game" (page 53).
- Write code to send data to other participants. See "Sending Data to Other Players" (page 56).
- Implement a match delegate:
  - 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 55).
  - The delegate receives the data that other players send. See "Receiving Data from Other Players" (page 56).
  - If players get disconnected while your game is running, your 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 57).
- Optionally, add support for voice between the match participants. See "Adding Voice Chat to a Match" (page 59).

Designing Your Network Game

Your application relies on a `GKMatch` object to exchange data with other participants in the match. The `GKMatch` class does not define the format 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 unreliable network requires more effort to deliver packets. Reliable messages also have the advantage that they are delivered to the destination in the order they were sent.
- Unreliable messages may never reach their destination or may be delivered out of order. Unreliable transmissions are most useful for real-time transactions where a delay in transmission means the data would be no loner useful. For example, if your game is transmitting position and velocity information for a dead-reckoning algorithm, a delayed transmission means the position is badly out of date. Instead, send a new message with an updated position.

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 (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.

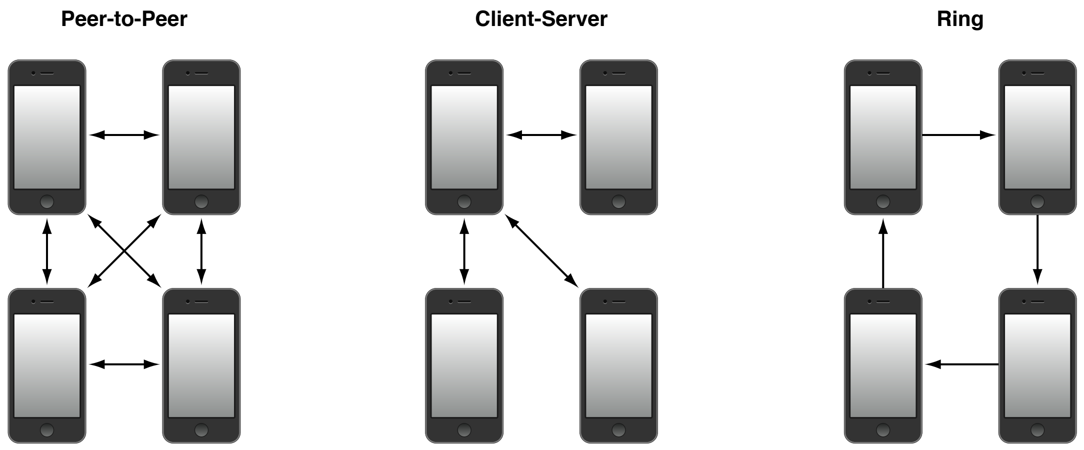
Be mindful that your network data is being transmitted across servers and routers that are out of your control. Your messages are subject to inspection and modification by other devices on the network. When your application receives network data from other participants, it should treat that data as *untrusted* data, and validate it 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:

- Game Kit does not provide a mechanism to identify the type of message that was received. Your message format should include a way to differentiate between different kinds of messages. For example, you might create an enumerated type that identifies different kinds of messages. The first bytes in every message would start with this enumerated type.
- Send message 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 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.
- Unreliable messages should be no larger than 1000 bytes in size.
- Reliable messages should be no larger than 87 kilbytes in size.
- Only send messages to the participants that need the message. For example, if your game has two different teams, team-related messages should only be sent 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 55) 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 only transmits to or from the host, you can halve the number of connections. A ring architecture only allows devices to forward network packets to the next device, but further reduces the number of connections. Each topology has 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



- Networks are an inherently unreliable medium of communication. This means that a participant can be disconnected at any time while your game is running. Your game needs to handle disconnection messages. For example, if you implemented your game using a client-server architecture, your game might want to nominate a new device to take over being the server.

Starting the Match

When Game Kit delivers a `GKMatch` object to your game, the connections to other players in the game have not yet been established. The `playerIDs` array can be empty if no other players are connected, or it may hold a subset of players that have already been connected. Your application must wait until all players are connected before starting the match. To determine how many players are waiting to join the match, your application 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 55) shows the skeleton for an implementation of your `match:player:didChangeState:` method. In this example, the match delegate relies on its own `matchStarted` property to store whether the match is already in progress. If the match has not started and the count of expected players reaches zero, it executes its own code to start the match. In your application, 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 to other match participants, you create a message and encapsulate it in a `NSData` object. You can send this message 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 (page 56) shows how an application might send a position update to the other participants. It fills a struct 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
    }
}
```

If the method returns without setting an error, then the data was queued and will be sent when the network is available.

Receiving Data from Other Players

When the match receives data sent by another participant, it delivers it to your match delegate's `match:didReceiveData:fromPlayer:` method. Your implementation of this method must decode the data and use it to update its game state.

```
- (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 your application is ready to leave a match, it should call the match object's `disconnect` method. In addition, a player may also be disconnected if their device does not respond for a period of time. When a player disconnects from the match, the other participants in the match are notified through their match delegate.

Adding Voice Chat to a Match

A significant advantage to using the `GKMatch` class is that it provides built-in support for in-game voice. You can create one or more separate voice chat **channels**, each containing some of the players connected to a match. When a player speaks into a channel, only the participants connected to the same channel can hear it.

Voice chat is only available to participants of the match that have wi-fi connections.

Checklist for Adding Voice Chat to a Match

Once you've got a match, and the players are connected, you can create one or more voice channels. Follow these steps:

- Decide how many channels your application needs. For example, a free-for-all game might create a single channel for all participants. A team game might create a separate channel for each team, and another 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 60).
- Call the match object's `voiceChatWithName:` method once for each channel your application wants to create. The match returns a `GKVoiceChat` object that provides the channel. See "Creating Voice Channels" (page 60).
- Call the voice chat object's `start` method when you want to activate the channel. See "Starting and Stopping Voice Chat" (page 60).
- When the player wants to speak into a channel, enable the microphone for that channel. If your application creates multiple channels, only one channel may own the microphone at one time. See "Enabling and Disabling the Microphone" (page 61).
- 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 the Voice Chat" (page 61).
- Decide whether your application supports a push-to-talk model or whether it continuously samples 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 whether the player's voice is being transmitted.
- Optionally, implement an update handler to be called when a player connects or disconnects, or start or stops speaking. See "Implementing a Player State Update Handler" (page 62).

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 allows for both playing and recording sounds. If your game provides other sound effects, you probably already have an audio session. Listing 7-1 (page 60) shows the code necessary to create an audio session that allows the microphone to be used.

Listing 7-1 Setting a play and record audio session

```
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 define a particular channel. This is done through a **channel name**. A channel name is a string defined by your application that uniquely names the channel. When two participants join a channel with the same name, they are automatically connected to the same voice chat.

Listing 7-2 provides code that calls `voiceChatWithName:` twice. The first creates a team channel, the second creates a global channel. Both channels are returned to your application autoreleased, so the code retains each one.

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

After joining a channel, voice data is not sent or received until the voice channel 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 connects to other participants in the channel provided 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 or if you want to temporarily turn off of a channel, you stop the chat:

```
[teamChannel stop];
```

An advantage to stopping the channel (rather than simply muting its participants) is that other participants are not required to send data to this participant. This conserves network bandwidth leaving more bandwidth available to your application.

Enabling and Disabling the Microphone

When you want the participant to be able to speak, you enable the microphone. Depending on your game, you may want to enable it 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 own 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 the Voice Chat

Your application can control the voice channel in two different ways. First, it can set the overall volume level for the channel 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 application intends to use this, 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 the player, you make the same call, passing `NO` instead.

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

Implementing a Player State Update Handler

Your application implements an update handler when it wants to see changes in a player's status. The handler is a block 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 62) shows an implementation of a player state update handler.

Listing 7-3 Setting a player state update handler

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

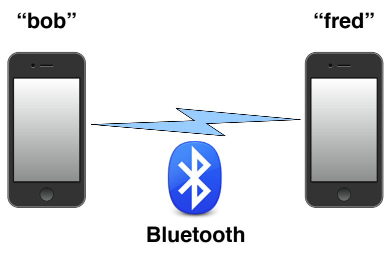
Peer-To-Peer Connectivity

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

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

Requiring Peer-to-Peer Connectivity

If your application requires Peer-to-Peer Connectivity, you want to ensure that only users whose devices support it can purchase and download your application. To do this, you add the peer-peer key to the list of required device capabilities found in your application's Info.plist file. See "iTunes Requirements" in *iOS Application Programming Guide*.

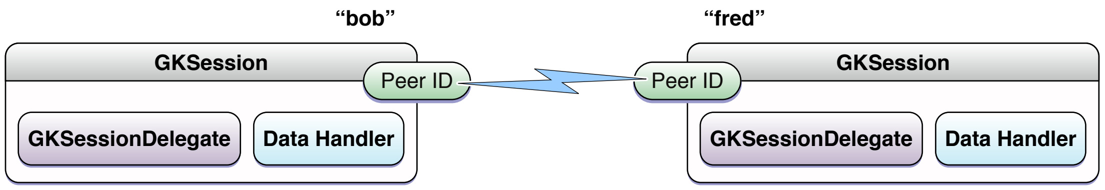
Sessions

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

Figure 2 Peer IDs are used to interact with other peers



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

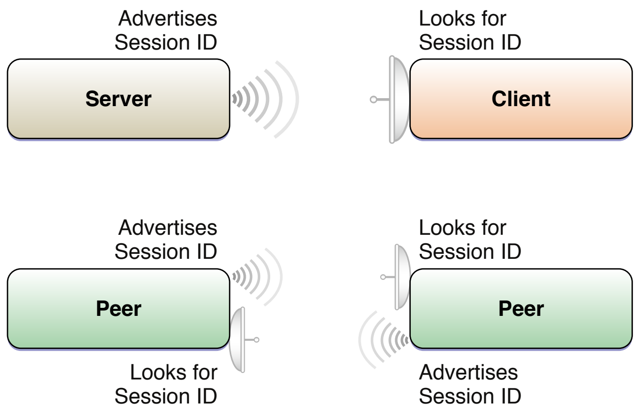
Discovering Other Peers

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

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

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

Figure 3 Servers, clients, and peers



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

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

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

Implementing a Server

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

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

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

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

Connecting to a Service

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

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

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

Exchanging Data

Peers connected to the session can exchange data with other connected peers. Your application sends data to all connected peers by calling the `sendDataToAllPeers:withDataMode:error:` method or to a subset of the peers by calling the `sendData:toPeers:withDataMode:error:` method. The data is an arbitrary block of memory encapsulated in an `NSData` object. Your application can design and use any data formats it wishes for its data. Your application is free to create its own data format. For best performance, it is recommended that the size of the data objects be kept small (under 1000 bytes in length). Larger messages (up to 87 kilobytes) may need to be split into smaller chunks and reassembled at the destination, incurring additional latency and overhead.

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

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

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

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

Disconnecting Peers

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

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

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

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

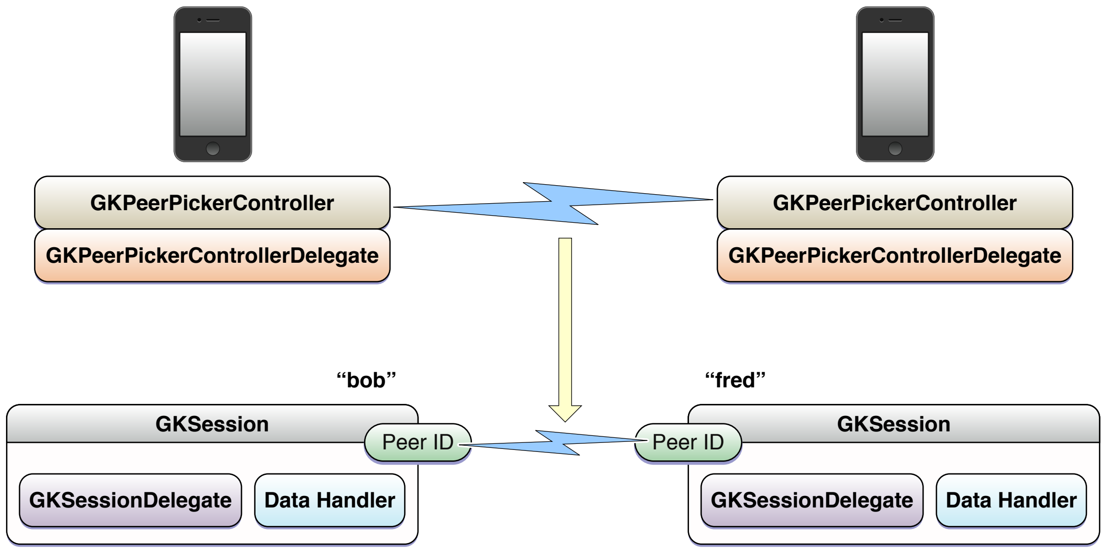
Cleaning Up

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

The Peer Picker

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

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



Configuring the Peer Picker Controller

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

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

Important: The peer picker only creates Bluetooth connections. If your application provides Internet connections, when the user selects an Internet connection, your application must dismiss the peer picker and present its own user interface to configure the Internet connection.

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

Displaying the Peer Picker

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

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

Finding Peers with Peer Picker

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

After you've read this article, you should read "Working with Sessions" (page 73) to see what your application can do with the created session.

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

1. Create and initialize a `GKPeerPickerController` object.

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

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

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

3. Configure the allowed network types.

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

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

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

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

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

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

```
    return session;
}
```

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

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

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

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

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

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

```
[picker show];
```

Working with Sessions

This article explains how to use a `GKSession` object that was configured by the peer picker. For more information on how to configure the peer picker, see "Finding Peers with Peer Picker" (page 71).

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

To use a session inside your application, first follow the steps found in "Finding Peers with Peer Picker" (page 71), then continue here.

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

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

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

2. Send data to other peers.

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

3. Receive data from other peers.

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

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

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

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

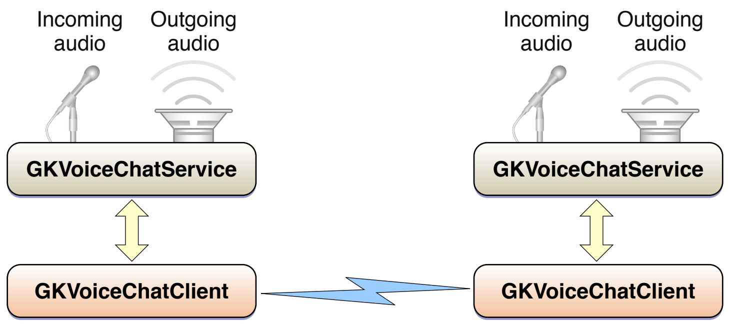
In-Game Voice

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

Figure 1 In Game Voice



Configuring a Voice Chat

Participant Identifiers

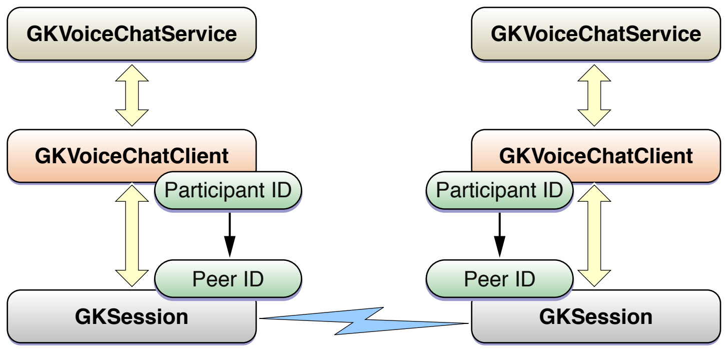
Each participant in a voice chat is identified by a unique **participant identifier** string provided by your client. The format and meaning of a participant identifier string is left to your client to decide.

Discovering other Participants

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

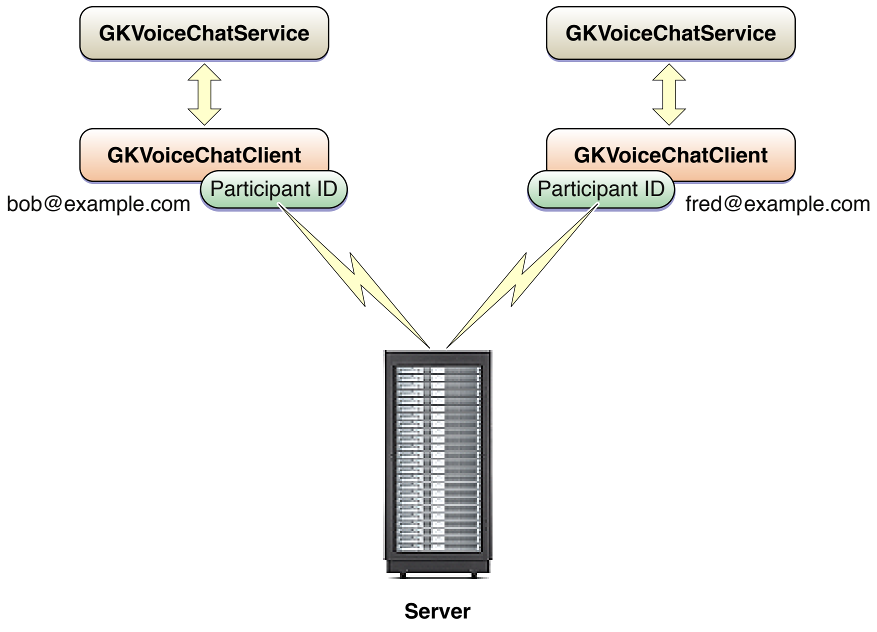
For example, if your application is already connected to another device through a GKSession object (see "Peer-to-Peer Connectivity" (page 65)), then each peer on the network is already uniquely identified by a peerID string. The session already knows the peerID string of the other participant. The client could reuse each peer's ID as the participant identifier and use the session to send and receive data, as shown in Figure 2

Figure 2 Peer-to-peer–based discovery



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

Figure 3 Server-based discovery



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

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

Real-time Data Transfer

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

Starting a Chat

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

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

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

Disconnecting from Another Participant

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

Controlling the Chat

Once the participants are connected, speech is automatically transmitted between the two iOS 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 retrieve microphone data. Similarly, your application can monitor the other participant by setting `outputMeteringEnabled` to `YES` and reading the `outputMeterLevel` property. To improve application performance, your application should only enable metering when it expects to read the meter levels of the two participants.

Adding Voice Chat

Voice chat is implemented on top of a network connection provided by your application. The following example uses a `GKSession` object to provide a network to the client. For more information on `GKSession` objects, see "Peer-to-Peer Connectivity" (page 65). To implement voice chat, perform the following steps:

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

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

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

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

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

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

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

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

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

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

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

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

5. Attach the client to the voice chat service.

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

6. Connect to the other participant.

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

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

7. Implement optional client methods.

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

Document Revision History

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

Date	Notes
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.