NP Toolkit Library Overview

PlayStation®Vita Programmer Tool Runtime Library Release 3.50

Table of Contents

1 Library Overview

Features

The NP Toolkit is a library for PlayStation®Vita that allows applications to easily implement key PlayStation™Network features. The main purpose of the library is to provide an easier way of creating and managing *session* information for the application. The main functions provided by the library allow you to:

- Initialize and terminate the NP Toolkit library
- Get basic network information, trigger PlayStation™Network sign in, and test a user's bandwidth
- Rank user scores on the ranking boards
- Matchmake between PlayStation™Network users
- Messaging between PlayStation™Network users
- Trophy unlocking and Information
- Integrate in-game commerce

System Resources

The library uses the system resources shown in Table 1.

Table 1 System Resources

Resource	Description
Thread	One thread between calling `init()` and returning from `terminate()`. Priority is specified by the application.
Processor time	Checks system callbacks and polls services 30 times per second.

Embedding into a Program

Include `np_toolkit.h` in your source program files (various header files will be automatically included as well).

After building your program, link `libSceNpToolkit.a` and `libSceNpToolkitUtils.a`.

Interfaces

NP Toolkit provides functionality to the services of the PlayStation™Network via these key interfaces:

- Network Information
- Friends
- Presence
- Ranking
- Matching
- Messaging
- Trophy
- Authentication
- Commerce

Sample Programs

A SampleMenu program is provided to show basic usage of NP Toolkit. This program is located at `%SCE_PSP2_SAMPLE_DIR%\sample_code\network\api_np_toolkit\np_toolkit_sample`.

Related Documentation

Refer to the following documents for information about the functions that NP Toolkit encapsulates:

- *Network Overview*

2 Using the Library

Basic Procedure

This section describes the basic procedure for using the NP Toolkit library. The processing flow is as follows:

(1) Set an object variable of the `sce::Toolkit::NP::Parameters` class as a parameter.

(2) Initialize the library via `sce::Toolkit::NP::Interface::init()`.

(3) Perform any required network activity.

(4) Terminate the library via `sce::Toolkit::NP::Interface::terminate()`.

Details about these steps are as follows:

(1) Prepare.

First, prepare an instance of the `sce::Toolkit::NP::Parameters` class, and at least specify a `NpToolkitCallback()` function to receive event notifications and a `CommunicationId` object.

(2) Initialize the library.

Call `sce::Toolkit::NP::Interface ::init()` and initialize the library. Specify the `sce::Toolkit::NP::Parameters` object that you set in step (1). The above API also loads and initializes the Application Utility Library.

(3) Perform any activity requiring PlayStation™Network.

See the following sections on possible activities.

(4) Terminate the library.

When you no longer need the library, call `sce::Toolkit::NP::Interface::terminate()`.

Table 2 Main APIs Used in the Basic Procedure

API	Description
`sce::Toolkit::NP::Parameters`	Parameter class that is used to store settings for the callback, output, and IDs.
`sce::Toolkit::NP::Interface::init()`	Initializes the library.
`sce::Toolkit::NP::Interface::terminate()`	Terminates the library.

Procedure for Using Future<T>

Because NP Toolkit runs in its own thread and most of the key functionality requires network calls, the interfaces have been designed to be asynchronous by nature. Results are communicated using `Future<T>` objects, which wrap an instance of the type.

A Future object is effectively "locked" until the result is ready or until an error has occurred. There are three methods for checking whether such objects are locked.

- **Blocking.** This can be achieved by calling `Future::hasResult()` or `Future::hasError()`. This effectively provides the same semantics as a synchronous call.
- **Note**: `Future::waitFor()` should not be used.

```
Future<MyData> future;
Interface::getData(&future);
```

```
While(!future.hasResult() && ! future.hasError()) {
 if(future.hasResult() )
   future.get();
 else
   future.getError();
}
```

- **Checking for results.** This can be done by calling `Future::hasResult()` and `Future::hasError()`. If these are not yet true, then the data is still being processed.

```
static Future<MyData> s_future;

// myFunction() called somewhere
myFunction() {
  Interface::getData(&s_future);
}

// in some other part of code later in execution check for a result
if(s_future.hasResult() )
     s_future.get()
if(s_future.hasError() )
     s_future.getError()
```

- **Checking NPToolkitCallback.** Every function in the NP Toolkit interface returns a specific event type to the NP Toolkit callback function; these will indicate if the result is ready or whether there is an error. This is effectively when the related `Future<T>` will return true for `hasResult()` or `hasError()`.

```
The NP Toolkit callback function is called in the context of
sce::Toolkit::NP::Thread.

static Future<MyData> s_future;
Interface::getData(&s_future);

MyNpToolkitCallback(Event)
{
    if(Event.event == myDataEvent)
    if( s_future.hasResult() )
        s_future.get();   // defer actual processing of the data until
                          outside the callback
}
```

3 Network Information

NP Toolkit provides an interface for retrieving network information.

Bandwidth Information

Call `getBandwidthInfo()` to obtain bandwidth information. For the argument *`bandwidthInfo`*, specify the *`Future`* variable for receiving the obtained results.

The following code example shows the procedure for obtaining bandwidth information.

```
Future<NpUtilBandwidthTestResult> bandwidthInfo;
int ret =
sce::Toolkit::NP::NetInfo::Interface::getBandwidthInfo( &bandwidthInfo );

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
while(!bandwidthInfo->hasResult() && !bandwidthInfo->hasError())
{
    // Waiting
}

printf("Upload bps = %lf\n", bandwidthInfo->get()->upload_bps);
printf("Download bps = %lf\n", bandwidthInfo->get()->download_bps);
```

Note: The bandwidth test can take quite a long time to complete (in the order of seconds) depending on the connection, so it *not* recommended that you wait for the test to complete by polling `hasResult()`. Instead you should call `getBandwidthInfo()`, and then continue processing after receiving the `netInfoGotBandwidth` event in the NP Toolkit callback.

Note: The bandwidth test feature will fail if the internet connection is behind a NAT type of 3 because the protocol does not support this network configuration.

Basic Network Information

Call `getNetInfo()` to obtain basic information about the users network, such as online status, IP address and NAT information. For the argument, *`state`*, specify a pointer to the Future variable for receiving the obtained results.

The following code example shows the procedure for obtaining basic network information.

```
int ret;
Future<sce::Toolkit::NP::NetStateBasic> state;

ret = sce::Toolkit::NP::NetInfo::Interface::getNetInfo(&state);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

Detailed Network Information

Call `getNetInfoDetailed()` to obtain all information about the user's network. For the argument, `state`, specify a pointer to the *`Future`* variable for receiving the obtained results.

The following code example shows the procedure for obtaining detailed network information:

```
int ret;
Future<NetInfoService::NetStateDetailed> state;

ret = sce::Toolkit::NP::NetInfo::Interface::getNetInfoDetailed(&state);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

Note: In most cases, the detailed network information is only really useful for debugging.

Signing in to the PlayStation™Network

Call `psnLoginDialogStart()` to start the Network Check Dialog. This allows the user to sign in to the PlayStation™Network if they are offline. The dialog will close automatically when processing has completed.

The following code example shows the procedure for starting the PlayStation™Network sign-in dialog.

```
int ret = psnLoginDialogStart();

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

You must be calling `sceCommonDialogUpdate()` to allow the log-in dialog to display.

List of Functions

Table 3 Network Related Functions

Function	Description
`getBandwithInfo()`	Measures bandwidth information.
`getNetInfo()`	Gets basic network information.
`getNetInfoDetailed()`	Gets detailed network information (this information is mostly useful for debugging).
`psnLoginDialogStart()`	Starts the Network Check Dialog and prompts the user to sign in to the PlayStation™Network.

4 User Profile

Online ID

Call `getOnlineID()` to obtain the users Online ID. For the `onlineId` argument, specify a pointer to the Future variable which will receive the result. The Online ID is the public identifer for a user in the PlayStation™Network.

The following code example shows the procedure for obtaining the user's Online ID.

```
int ret;
Future<SceNpOnlineId> onlineId;

ret = UserProfileInterface::getOnlineID(&onlineId);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

NP ID

Call `getNPID()` to obtain the users NP ID. For the `npid` argument, specify a pointer to the Future variable which will receive the result. The NP ID is used by the system utilities of the PlayStation™Network for identifying the user.

The following code example shows the procedure for obtaining the users NP ID.

```
int ret;
Future<SceNpId> npid;

ret = UserProfileInterface::getNPID(&npid);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

Avatar URL

Call `getAvatarUrl()` to obtain the URL for the user's PlayStation™Network avatar. For the `avatarUrl` parameter, specify a pointer to the Future variable which will receive the result.

The following code example shows the procedure for obtaining a URL for the user's avatar.

```
int ret;
Future<SceNpAvatarUrl> url;

ret = UserProfileInterface::getAvatarUrl(&url);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

Cached User Information

Call `getCachedUserInfo()` to obtain the user data that is stored on the internal HDD. For the `userInfo` parameter, specify a pointer to the Future variable which will receive the current user's information.

The following code example shows the procedure for obtaining the user's cached information.

```
int ret;
Future<SceNpManagerCacheParam> info;

ret = UserProfileInterface::getCachedUserInfo(&info);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

User's Country Information

Call `getCountryInfo()` to obtain the user's country information. This comprises of a two character ASCII country code defined in ISO 3166-1 and the user's PlayStation™Network language (`CELL_SYSUTIL_LANG_XXX`). For the `info` argument, specify a pointer to the Future variable which will receive the result.

The following code example shows the procedure for obtaining the user's country information.

```
int ret;
Future<UserProfileService::CountryInfo> info;

ret = UserProfileInterface::getCountryInfo(&info);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

Parental Control and Content Restrictions

Call `getParentalControlInfo()` to obtain the user's age and parental control restrictions. For the `info` argument, specify a pointer to the Future variable which will receive the result.

The following code example shows the procedure for obtaining the user's parental control information.

```
int ret;
Future<UserProfileService::ParentalControlInfo> info;

ret = UserProfileInterface::getParentalControlInfo(&info);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

List of Functions

Table 4 User Profile Related Functions

Function	Description
`getOnlineID()`	Gets the user's Online ID.
`getNPID()`	Gets the user's NP ID.

Function	Description
`getAvatarUrl()`	Gets the URL of the user's avatar.
`getCachedUserInfo()`	Gets the user information that was cached by the system.
`getCountryInfo()`	Gets the user's country code and language.
`getParentalControlInfo()`	Gets parental control information such as content and chat restrictions.
`getPlatform()`	Gets the current platform the application is running on.

5 Friends

NP Toolkit provides a simple interface for retrieving friend information as well as adding new friends.

Getting a Friends List

Call `getFriendslist()` to obtain friend information. For the `friendsList` parameter, specify a pointer to the Future variable which will receive the results.

The NP Toolkit library maintains the friend list automatically and provides notification through the NP Toolkit callback when it has changed. Friends "presence information" is also contained within the `Friend` structure, so an application can identify if a friend is online and playing the same game.

The following code example shows the procedure for obtaining friend information.

```
int ret;
Future<std::vector<Friend> > friendslist;
FriendInfoRequest request;

request.flag = SCE_TOOLKIT_NP_FRIENDS_LIST_ALL;
request.userInfo.userId = userId;

ret = FriendsInterface::getFriendslist(&friendslist,&request);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
// Error handling
}
```

The NP Toolkit library also provides the functionality to store a friends list in the NP Toolkit library cache.

Note: In order to keep backwards compatibility with other platforms, the function `getFriendslist(sce::Toolkit::NP::Utilities::Future<Friend>*)` has also been provided. This function will be deprecated and should be avoided.

Getting a List of Blocked Users

Call `getBlockList()` to obtain a list of the users which are currently blocked by the user. For the `blockList` parameter, specify a pointer to the Future variable which will receive the results.

The NP Toolkit library maintains the block list after the application has retrieved the block list once from the PlayStation™Network server. To request the cached results, specify `SCE_TOOLKIT_NP_BLOCK_LIST_CACHED` as a parameter. An error will be returned if the cache is not found. If `SCE_TOOLKIT_NP_BLOCK_LIST_CACHED` is not specified, the library will make a request to the PlayStation™Network server to retrieve the block list. It is recommended that once the blocklist is retrieved, the application requests cached results from the NP Toolkit library.

The following code example shows the procedure for obtaining a list of the users which are currently blocked by the user :

```
int ret;
Future< BlockList > blocklist;
BlockedListInfoRequest request;

request.userInfo.userId = userId;
request.flag = SCE_TOOLKIT_NP_BLOCK_LIST_CACHED;// only call once the block is
retrieved
ret = FriendsInterface::getBlockList(&blocklist,&request);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
```

```
        {
        // Error handling
        }
```

List of Functions

Table 5 Friends Related Functions

Function	Description
`getFriendslist()`	Gets a list of the user's friends.
`getBlockList()`	Gets a list of users which are blocked.

6 Presence

NP Toolkit provides an interface for setting a user's presence.

Setting the Current User's Presence

To set a user's presence, call `setPresence()`, which takes a pointer to a `sce::Toolkit::NP::PresenceDetails` object. This structure contains a status member that has a length of 128 characters. Use this member to set a user's current stage, location, or any other in-game information of interest to the user's friends. The structure also contains two other members: a *`data`* member for optional binary data and a *`size`* member for the size of the data. The former can be application-specific binary data, which can be used, for example, to hold session information about an individual user and so forth. The maximum length the binary data can be is 128 bytes.

It is also possible to delete the presence status and binary data. To delete the game presence status, set `PresenceDetails::status` to an empty string. Delete the game presence binary data by specifying a size of 0 for `PresenceDetails::size`.

When setting/deleting the presence status or presence binary data, the application will need to set the `PresenceDetails::presenceType` member to either `SCE_TOOLKIT_NP_PRESENCE_STATUS` or `SCE_TOOLKIT_NP_PRESENCE_DATA`. These are mutually exclusive flags so it is not possible to OR these two flags.

In addition to the structure parameter, `setPresence()` takes a second parameter, *`async`*, which is used to indicate the mode (see the "Note" below).

The following code example shows the procedure for setting a user's presence.

```
sce::Toolkit::NP::PresenceDetails presDetails;
memset(&presDetails, 0x00, sizeof(presDetails));

presDetails.status = "Updating user's presence: stage 1!";
presDetails.userInfo.userId = userId;
presDetails.presenceType = SCE_TOOLKIT_NP_PRESENCE_STATUS;

int ret = sce::Toolkit::NP::Presence::Interface::setPresence(&presDetails,
async);
if(ret < 0)
// Error handling
```

The following code example shows the procedure for deleting a user's presence.

```
sce::Toolkit::NP::PresenceDetails presDetails;
memset(&presDetails, 0x00, sizeof(presDetails));

presDetails.userInfo.userId = userId;
presDetails.presenceType = SCE_TOOLKIT_NP_PRESENCE_STATUS;

int ret = sce::Toolkit::NP::Presence::Interface::setPresence(&presDetails,
async);
if(ret < 0)
// Error handling
```

Note: This call can be made in synchronous and asynchronous mode. It is generally preferable to make the call in asynchronous mode because you do not have to wait for anything to come back except events that indicate the failure or success of the call.

Getting the Presence of a Friend of the User

To get the presence of a friend of the user, call `getPresence()`, which takes a pointer to a `PresenceRequest` object. The application can specify the type of presence information to retrieve, such as Platform Presence, In Context Presence and Primary Presence, by using the *`presenceType`* member in the `PresenceRequest` object.

The presence information will be retrieved through the `PresenceInfo` Future object. On successful completion of the API (in the case of a synchronous call) or when a `presenceGotInformation` event is triggered in the NpToolkitCallback (in the case of an asynchronous call), the application can get the information using the `getResult()` method of the Future object. If an attempt is made to retrieve a user who is not a friend , the API will return an error code.

The following code example shows the procedure for getting the presence of a friend of the user:

```
sce::Toolkit::NP::Utilities::Future<PresenceInfo> s_presenceInfo;

SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&userId);
if( ret < 0 ) {
    // Error Handling
    //return
}
sce::Toolkit::NP::PresenceRequest presenceRequest;
memset(&presenceRequest,0,sizeof(presenceRequest));

strncpy(presenceRequest.onlineId.data,"SteveHd",strlen("SteveHd"));
presenceRequest.presenceType = SCE_TOOLKIT_NP_PRESENCE_TYPE_PLATFORM_INFO;
presenceRequest.userInfo.userId = userId;

sce::Toolkit::NP::Presence::Interface::getPresence(&presenceRequest,
    &s_presenceInfo,false);
if( ret < 0 ) {
    //Error Handling
}
```

List of Functions

Table 6 Presence Related Functions

Function	Description
`setPresence()`	Sets the user's presence information.
`getPresence()`	Gets the presence information of a friend of the user.

7 Ranking

NP Toolkit provides an interface for registering the current user's score as well as retrieving a friend's ranking information relating to the current game and retrieving a user-specified range of scores.

Note: Except for `registerCache()`, all of the ranking functions have deprecated versions available in addition to the new functions to maintain backwards compatibility. All of the following examples are based on the new functions.

Registering a Score

To register a score, call `registerScore()` and pass in three parameters:

- `RegisterScoreRequest* scoreToRegister`
- `Future<TempRank>*` *`tempRank`*
- `bool` *`async`*

The *`scoreToRegister`* parameter provides the information for registering a user's score. This includes the ID of the user making the request, the ID of the board to register the score in and the score itself. Optionally, game information in a binary format and a comment to go along with the score can also be included. NP Toolkit takes server load into consideration if `registerCache()` is called by minimizing the server calls; to do this, NP Toolkit will check the score and the board ID against the cache before it makes any server calls. It will make a server call to register the score only if the score is higher than the one in cache. This is also of course dependent on the board configuration: If it is set up so that latest score is always registered, then score filtering will not be carried out before making a server call.

In asynchronous mode, NP Toolkit will handle polling to see when the result of the server call has come back. The library will also handle transaction IDs; these transaction IDs are created if the NP Toolkit needs to make ranking server calls, such as registering a score or retrieving scores from the ranking server. A maximum of 32 transaction IDs can exist at any one time.

The following code example shows the procedure to register user's score

```
sce::Toolkit::NP::Utilities::Future<sce::Toolkit::NP::TempRank> tempRank;
sce::Toolkit::NP::RegisterScoreRequest scoreToRegister;

memset(&scoreToRegister, 0x00, sizeof(scoreToRegister));

scoreToRegister.gameInfo.infoSize = 64;
scoreToRegister.boardId = 1;          // Board number
scoreToRegister.score = 999919999;  // Score you want to register
memcpy(&scoreToRegister.comment, "testing rank comment",
SCE_NP_SCORE_COMMENT_MAXLEN);

ret = Ranking::Interface::registerScore(&scoreToRegister, & tempRank, false);
if(ret < 0)
{
// Error handling here
}
if (tempRank ->hasResult() && (tempRank ->hasError() == SCE_TOOLKIT_NP_SUCCESS))
{
    // print temprank here
}
```

Note: Every time the ranking server is pinged in debug mode, NP Toolkit will work out the frequency of calls that the application has made to the server and will output warnings if on average the application made more than 1 server call in less than 5 minutes.

Displaying a Range of Ranks

To retrieve a range of ranks for displaying purposes, call `displayRangeOfRanks()` and pass in three parameters:

- `RangeOfRanksRequest * rangeRequest`
- `Future<sce::Toolkit::NP::RankInformation> * score`
- `bool async`

The *rangeRequest* parameter provides the information to set up the request correctly. This includes the ID of the user making the request , the ID of the board to retrieve the ranks from, the first rank in the range and the number of ranks in the range. The *score* parameter is where all the retrieved ranks are set after NP Toolkit has successfully retrieved this information from the ranking server. A maximum of `SCE_TOOLKIT_NP_MAX_RANGE(30)` can be retrieved at one time. The policy is to retrieve enough ranks for displaying on a user's screen; more than 30 would be difficult for users to see. Also specify from what board ID you want to retrieve the ranks and specify how many ranks by providing the starting rank and how many ranks from that starting rank.

If `registerCache()` is called, internally NP Toolkit will create 4 (`SCE_TOOLKIT_NP_MAX_NUM_BOARD`) tables of caches for range of ranks, each holding x lines of ranks that was specified in `registerCache()`. A maximum of 4 different boards can be cached at any given time. By using modulus calculation, board numbers can be looped to cache board IDs of over 4; this same technique can be applied to lines of ranks that are x amount of lines specified when `registerCache()` function was called. Every time this function is called, cache will be checked first to see if these ranges of ranks exists in there. Only make server calls to retrieve these ranks if they do not exist in cache.

The following code example shows the procedure for retrieving range of ranks for display purposes.

```
const int scoreRange = 30;   // n ranks, starting from startRank, to be retrieved

Future<sce::Toolkit::NP::RankInformation> rangeScore;

RangeOfRanksRequest request;
request.boardId = 0;        // The ID of the board to retrieve rankings from
request.startRank = 1;      // Starting rank that you want to retrieve
request.range = scoreRange;

ret = Ranking::Interface::displayRangeOfRanks(&request, rangeScore, true);
if(ret < 0){
        // Error handling
}
if(rangeScores.hasResult() && rangeScores->hasError() ==
    SCE_TOOLKIT_NP_SUCCESS){
    int numRankReturned = rangeScores->get()->rankReturned;
    for (int rank = 0; rank < numRankReturned; rank++) {
         printf("%d. ", rangeScores->get()->rankData[rank].rank);
         printf("%s ", rangeScores->get()->rankData[rank].npId.handle.data);
         printf("%ull ", rangeScores->get()->rankData[rank].scoreValue);
         printf("%s\n", rangeScores->get()->comment[rank].utf8Comment);
    }
}
```

Note: Every time the ranking server is pinged in debug mode, NP Toolkit will work out the frequency of calls that the application has made to the server and will output warnings if on average the application made more than 1 server call in less than 5 minutes.

Displaying the Ranks of a User’s Friends

To retrieve some friends’ ranks for displaying purposes, call `displayFriendRank()` and pass in three parameters:

- `FriendRankRequest` *`request`*
- `Future<sce::Toolkit::NP::FriendsRankInformation> *` *`friendRank`*
- `bool` *`async`*

The r*`equest`* parameter provides the information to set up the request correctly. This includes the ID of the user making the request and the ID of the board to retrieve the ranks from. The *`friendRank`* parameter is where all the retrieved ranks are set after NP Toolkit has successfully retrieved this information from the ranking server. A maximum of `SCE_TOOLKIT_NP_MAX_RANGE(101)` can be retrieved at one time, this includes current user plus maximum number of current user’s friend, which is 100. If `registerCache()` is called, internally NP Toolkit will cache retrieved friend rank for a single board.

The following code example shows the procedure for retrieving the ranks of a user’s friends for display purposes.

```
Future<sce::Toolkit::NP::FriendsRankInformation> friendRank;

SceNpScoreBoardId boardId = 1;

FriendRankRequest request;
request.boardId = boardId;        // Board number
ret = Ranking::Interface::displayFriendRank (&request,
                                             friendRank,
                                             false);

if(ret < 0){
// Error handling
}

if(friendRank.hasResult() && friendRank ->hasError() ==
SCE_TOOLKIT_NP_SUCCESS){
  int maxNumberFriends = friendRank->get()->numFriends;
  for (int friendIdx = 0; friendIdx < maxNumberFriends; friendIdx++) {
    printf("Rank: %d. ", score->get()->rankData[friendIdx].rankData.rank);
    printf(" %ull ", score->get()->rankData[friendIdx].rankData.scoreValue);
    printf(" %s ",  score->get()->rankData[friendIdx]
                            .rankData.npId.handle.data);
    printf(" %s \n", score->get()->comment[friendIdx].utf8Comment);
  }
}
```

Displaying a User’s Rank

To retrieve a current user’s own rank or the specified rank of another user for displaying purposes, call `displayUserRank()` and pass in three parameters:

- `UserRankRequest*` *`request`*
- `Future<sce::Toolkit::NP::UserRankInformation>*` *`currentUserScore`*
- `bool` *`async`*

The *`request`* parameter provides the information to set up the request correctly. This includes the ID of the user making the request, the NP ID of the user to request the rank for, and the ID of the board to retrieve the rank from. The *`currentUserScore`* parameter receives the retrieved ranks after NP Toolkit has successfully retrieved this information from the ranking server. If *`npId`* is memsetted to `0x00`, then NP Toolkit will retrieve the ranking information for the current user; otherwise the ranking information of the user specified by *`npId`* will be retrieved.

The following code example shows the procedure for retrieving the current user's rank for display purposes.

```
Future<sce::Toolkit::NP::UserRankInformation> ownRank;
UserRankRequest request;
request.boardId = 1;
memset(&request.npId, 0x00, sizeof(request.npId)); /* Pass in an empty SceNpId,
this will retrieve the current user's rank by default. */

ret = Ranking::Interface::displayUserRank(&request, ownRank, true);
if (ret < 0) {
    printf("RankingInterface::displayFriendRank() failed, ret = 0x%x\n", ret);
    return;
    }
```

Registering the Cache

To use ranking service cache you will have to call `registerCache()` to allocate memory for the cache. There are 4 types of cache:

- Board Cache: caches board configuration information.
- Write Cache: caches registered scores
- Read Cache: caches range of scores retrieved via `displayRangeOfRanks()` function call
- Friend Cache: caches friend scores for a single score board.

For Board, Write and Read cache you can pass in number of lines of cache you would like to use. For Friend cache you can only pass in true or false to enable or disable it. Each line of caches will allocate the following amount of memory:

- Board Cache: each line is 24 bytes.
- Write Cache: each line is 289 bytes.
- Read Cache: each line is 416 bytes.
- Friend Cache: Total cache size is 40,024 bytes, which holds 101 lines is of rank information.

These are optional and you can choose which caches to use – by passing in 0 you will omit that specific cache.

The following code example shows the procedure for registering cache.

```
int boardLineCount = 8;
int writeLineCount = 20;
int rangeLineCount = 128;
bool friendCache = true;
ret = Ranking::Interface::registerCache(boardLineCount,
                                        writeLineCount,
                                        rangeLineCount,
                                        friendCache);

if(ret < 0){
// Error handling
}
```

Terminating the Ranking Service

To explicitly terminate ranking service, call `rankingTerm()`, this will unload ranking library, free up memory used for cache and executes the following function calls:

- `sceNpScoreDeleteTitleCtx()`
- `sceNpScoreTerm()`

Initializing the Ranking Service

To initialize the ranking service after calling `rankingTerm()` call `rankingInit()`, this will load the ranking library module and executes the following function calls:

- `sceNpScoreInit()`
- `sceNpScoreCreateTitleCtx()`

Note: You do not have to call this initially when the NP Toolkit library is initialised, only after if `rankingTerm()` was called. This is because ranking service will be initialized automatically first time ranking service is used.

List of Functions

Table 7 Ranking Related Functions

Function	Description
`registerScore()`	Registers the current user's score.
`displayRangeOfRanks()`	Retrieves a range of ranks for display purposes.
`displayFriendRank()`	Retrieves some friends' ranks from a specified board.
`displayUserRank()`	Retrieves the current user's rank.
`registerCache()`	Allocates a ranking service cache.
`rankingTerm()`	Unloads the ranking library and deallocates cached memory.
`rankingInit()`	Loads the ranking library and sets up the ranking service.

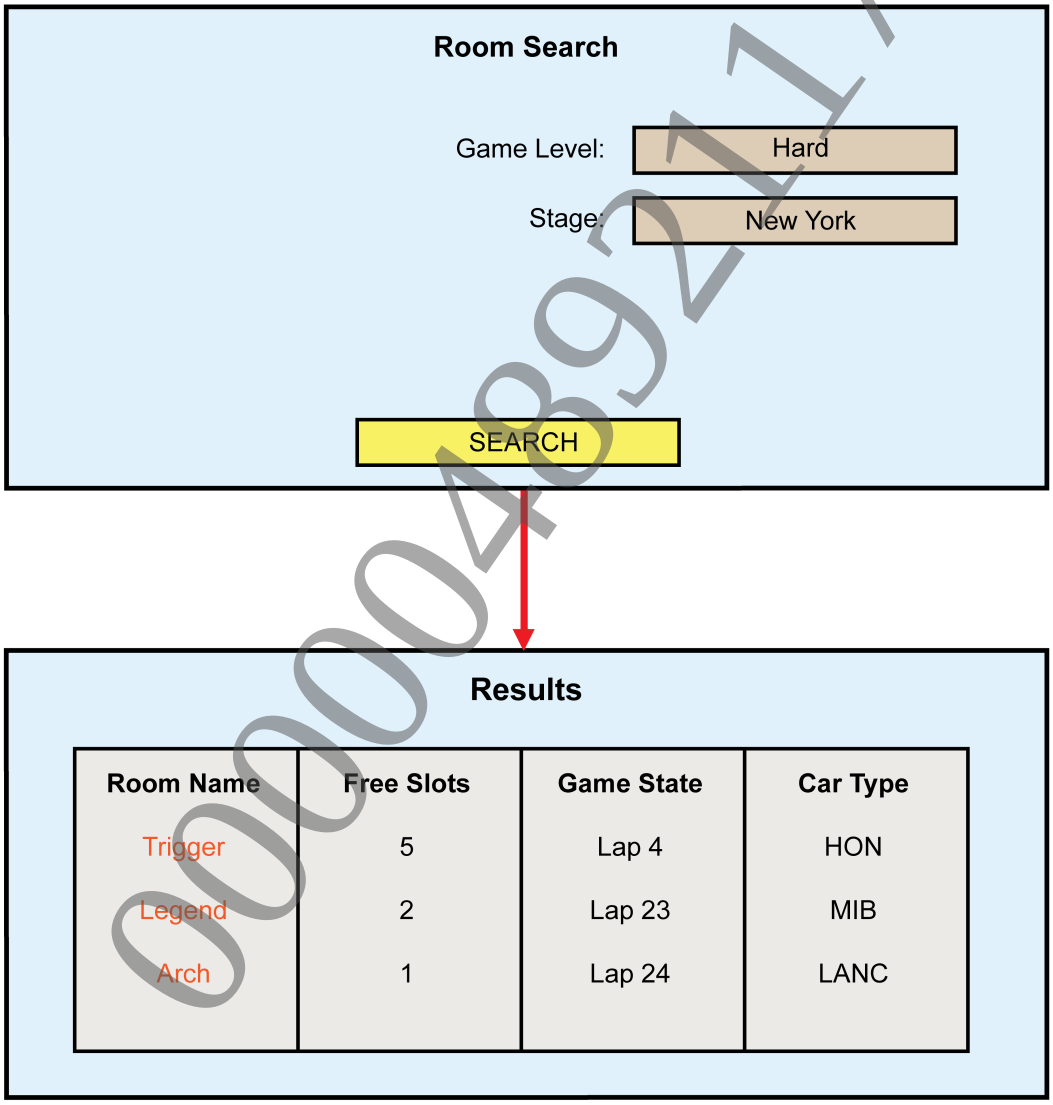
8 Matching

The NP Toolkit library provides interface functions to create, search, join, and update a matching room on the NP Matching Server. For development process on how to configure title to use Matching Service please refer to the "Development Process Section" in the *NP Matching 2 System Overview*.

What Are Session Attributes?

To appropriately use the NP Toolkit library matching interface, you must understand the concept of session attributes. Figure 1 and the following discussion explain the concept of session attributes and the procedure associated with it.

Figure 1 Examples of Session Attributes



Room Name	Free Slots	Game State	Car Type
Trigger	5	Lap 4	HON
Legend	2	Lap 23	MIB
Arch	1	Lap 24	LANC

In this figure, *Game Level* and *Stage* can be described as session attributes because they can be used to search for a session or to specify information about the session. Similarly, in the *Results* view of the diagram, *Game State* and *Car Type* can also be described as session attributes to represent internal room data.

The following sections provide more specific details about session attributes and their usage.

Session Attribute Categories

Each session attribute in your application must belong to one, and only one, of four categories (see Table 8). These attributes can be set or retrieved from a session using the library's matching interface function. They can be used to filter the session or set important details about the session.

Table 8 Session Attribute Categories

Session Attribute Category	Description
`SCE_TOOLKIT_NP_SESSION_SEARCH_ATTRIBUTE`	Session attribute will be used as search attribute.
`SCE_TOOLKIT_NP_SESSION_EXTERNAL_ATTRIBUTE`	Session attribute will be used as external room data.
`SCE_TOOLKIT_NP_SESSION_INTERNAL_ATTRIBUTE`	Session attribute will be used as internal room data.
`SCE_TOOLKIT_NP_SESSION_MEMBER_ATTRIBUTE`	Session attribute associated with the room member.

Session Attribute Value Types and Flags

Each session attribute will have a value associated with it, and it is necessary for the application to specify what the value type of the session attribute will be during registration.

Table 9 shows a list of flags that must be specified when registering a session attribute. Make sure that you do not specify multiple flags for a session attribute (this is true for any type of flag).

Table 9 Session Attribute Value-Type Flags

Session Attribute Value-Type Flag	Description
`SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_VALUE_INT`	Session attribute value will be type `int`.
`SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_VALUE_BINARY`	Session attribute value will contain binary data. When specifying this flag it is necessary to mention the maximum size of binary data using the `SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_MAX_SIZE_`*`XXX`* flag (see below).

If a session attribute value type is binary, the application also has to set the max-size flag for the session attribute. (The application should not set this flag if the value type is integer; in that case, the `SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_MAX_SIZE_`*`XXX`* flag is ignored, where *XXX* is the particular size.)

Table 10 shows a list of flags and the corresponding maximum size of data that can be assigned to a session attribute value.

Table 10 Session Attribute Max-Size Flags

Session Attribute Max-Size Flag	Maximum Size of Binary Data
`SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_MAX_SIZE_12`	12 bytes
`SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_MAX_SIZE_28`	28 bytes
`SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_MAX_SIZE_60`	60 bytes
`SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_MAX_SIZE_124`	124 bytes
`SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_MAX_SIZE_252`	252 bytes

After the session attributes that will be used for a session are decided, the application can register them as discussed in the following section.

Registering Session Attributes

Before using such data as session attributes, your application must first register them as session attributes. Your application should register session attributes only once; it is best to register them when the application is initialized. If your application specifies an unregistered session attribute when creating or searching for a session, then an appropriate error code is returned by the library indicating that the session attributes have not been registered. It is important for your application to note that the registration should be done before using any of the NP Toolkit library's matching interface functionality and that this registration should be done only once.

The following is an example of how session attributes must be registered:

```
sce::Toolkit::NP::RegisterSessionAttribute sessionAttribute;
memset(&sessionAttribute,0, sizeof(sessionAttribute));

strcpy(sessionAttribute.attribute,"LEVEL");
sessionAttribute.attributeType = SCE_TOOLKIT_NP_SESSION_SEARCH_ATTRIBUTE;
sessionAttribute.valueType = SCE_TOOLKIT_NP_SESSION_ATTRIBUTE_VALUE_INT;

Matching::Interface::registerSessionAttributes(&sessionAttribute,1 );
```

After session attributes are registered successfully, the attribute names can then be used to set the session attributes while creating, searching, or modifying a session

The remainder of this chapter provides specific usage details about session attributes.

Session Functionality

The main purpose of the library is to provide an easier way of creating and managing *session* information for the application.

To join a session, you must either create your own room or join a session that was created by another user. The information about this session is contained within the `SessionInformation` structure. For more information about the parameters of the structure, refer to the *NP Toolkit Library Reference*.

Creating a Session

To create a session, the application must execute `MatchingInterface::createSession()`. This function enables you to create a session and join a session. Specify the session parameters of the room by using the `CreateSessionRequest` structure.

Table 11 CreateSessionRequest Structure Members

Structure Member	Description	Specification	Change After Creation?
maxSlots	Total number of slots (maximum 64)	Required	Not possible
sessionFlag	Session flags	Optional	Not possible
slotsInformation	Information about the slots in current session	Optional	Not possible
sessionPassword	Password	Optional	Not possible
SessionAttributes	Session attributes	Optional	Possible

Code Example for Setting the createSessionRequest Parameter

The API `Matching::Interface::createSession()` function call is used to create a session with a specified parameter, as shown in the following example:

```
sce::Toolkit::NP::CreateSessionRequest createSessionRequest;
memset(&createSessionRequest, 0 ,
sizeof(sce::Toolkit::NP::CreateSessionRequest));
```

Specify Session Flags (optional)

To specify session flags, enable the flags by OR'ing them. To disable all the flags, specify 0.

```
//e -
createSessionRequest.sessionFlag =
(SCE_TOOLKIT_NP_CREATE_PASSWORD_SESSION |
SCE_TOOLKIT_NP_CREATE_HOST_MIGRATION_SESSION);
```

Specify Slot Information

The following example shows how to create a session with a capacity to hold 8 members:

```
createSessionRequest.maxSlots = 8;
```

If the session flag is set to `SCE_TOOLKIT_NP_CREATE_CUSTOM_SESSION`, the application can specify how many slots it wants to be reserved for private or public players. In the example below, the application reserves 4 slots for private players:

```
createSessionRequest.slotsInformation.reservedSlots = 4;
```

If the session flag is set to `SCE_TOOLKIT_NP_CREATE_SESSION_TYPE_PRIVATE`, only friends can join the game through invitation.

Specify Password

If the session flag is set to `SCE_TOOLKIT_NP_CREATE_PASSWORD_SESSION`, only the application can specify the password for the session. If the flag is not set, the session is not password protected even if the password is set.

```
strncpy(createSessionRequest.sessionPassword, "ROOMPSWD",
SCE_NP_MATCHING2_SESSION_PASSWORD_SIZE);
```

Specify Attributes for the Session

Specify the session attributes you want to set for the session that will be created. Make sure that the session attribute name that will be used has been registered using the `registerSessionAttributes()` function call. Specify a value consistent with the value type that was registered.

```
SessionAttributeRequest sessionAttribute;
memset(sessionAttribute,0,sizeof(sessionAttribute));
strcpy(sessionAttribute.attribute,"LEVEL");
sessionAttribute.attributeValue.attributeIntValue = 145;

createSessionRequest.numSessionAttributes = 1 ;
createSessionRequest.sessionAttributes = sessionAttribute;
```

Note: In the this case, make sure that `LEVEL` has been registered using the `registerSessionAttributes()` function.

Specify World ID

Specify a World ID if you want your application to create a session in a specific world. If `worldId` is set to 0, the library will randomly select a world and create a session in that world.

Specify Server ID

Specify a Server ID if you want your application to create a session on a specific server. If `servrId` is set to 0, the library will randomly select a server and create a session on that server. It is recommended that you specify 0 and let the library select a server for the application.

Creation Process

When setting the create request parameter, the application needs to pass in the request to the `Matching::Interface::createSession()`. If the request is successful, the function will return

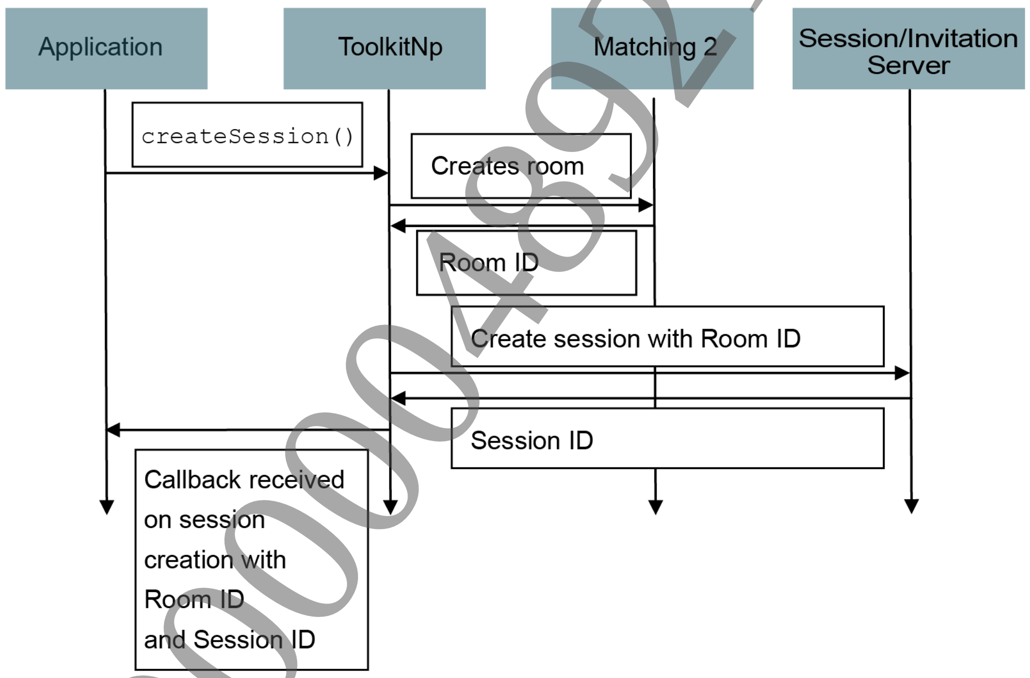
SCE_TOOLKIT_NP_SUCCESS. The application is notified by a callback event indicating that the request process was completed. To check whether the session was successfully created or if there was an error, the application must check the hasResult() or hasError() of the Future object. If a session is created successfully, the application can get the information of the session from the getData() interface function of the Future object.

```
static Future< sce::Toolkit::NP::SessionInformation> sessionInformation;

ret = Matching::Interface::createSession(&createSessionRequest,&
                                         sessionInformation);
if (ret < 0 ) {
printf("Unable to create a request for session creation 0x%x\n",ret);
}
```

Once the session is created, the library will internally register a session on the session server. The *npSessionId* param of the SessionInformation structure will contain the information about the session. If the *npSessionId* param is empty, it indicates the session failed to register on the session server. The error code can be checked in the *sessionServerState* param of the SessionInformation structure. In the current implementation, failure to register the session on the session server will result in success as the NpMatching2 room was actually created successfully. The process of above API is shown in Figure 2.

Figure 2


Application
ToolkitNp
Matching 2
Session/Invitation Server
createSession()
Creates room
Room ID
Create session with Room ID
Session ID
Callback received on session creation with Room ID and Session ID

Searching for Sessions

To join a session created by another user, it is necessary to obtain the session information of the session. This section describes how to perform a search for specific session or list of sessions.

Session Search Request

To search for a session, execute Matching::Interface::searchSessions(). Specify search-request parameters to this API function call. If the search is successful, a list of session information is returned to the application. The parameters to be specified in the request structure of Matching::Interface::searchSessions() are mentioned in the following sections.

Initialize the Search Request

```
sce::Toolkit::NP::SearchSessionsRequest searchRequest;
memset(&searchRequest , 0 ,
sizeof(sce::Toolkit::NP::SearchSessionsRequest));
```

Specify Search Flags

Table 12 Search Flags/Types of Sessions

Search Flags	Types of Sessions to be Searched
SCE_TOOLKIT_NP_SEARCH_FRIENDS_SESSIONS	Searches for a session hosted by a user's friends (note that this functionality is not provided in the current release).
SCE_TOOLKIT_NP_SEARCH_REGIONAL_SESSIONS	Searches for a session hosted by a user belonging to the same region or country.

```
//e- this will search for session hosted by the user belonging to same region
searchRequest. searchFlags  = SCE_TOOLKIT_NP_SEARCH_REGIONAL_SESSIONS;
```

Specify Filters

By default, a search will not return sessions that are hidden (a private session) or that are closed (when all slots are occupied). Specify a search filter (or filters) for attributes that were registered using `registerSessionAttributes()` and whose attribute category was set as `SCE_TOOLKIT_NP_SESSION_SEARCH_ATTRIBUTE`. Only a maximum of 8 search attributes of type integer can be used, and only 1 search attribute of type binary can be used. The size of the binary data cannot be more than 60 bytes.

```
sce::Toolkit::NP::SessionAttributeRequest searchFilter;
strcpy(searchFilter.attribute,"LEVEL");
searchFilter.attributeValue.attributeIntValue = 35;
searchFilter.searchOperator = SCE_NP_MATCHING2_OPERATOR_EQ;

searchRequest.searchFilters = searchFilter;
searchRequest.numSearchFilters = 1;
```

Specify Search Space

When searching for a session in a specific world or on a specific server, specify the *`worldId`* or *`serverId`* parameter of the session search request structure, respectively. If 0 is specified, the library will internally search for the session on a randomly selected search space.

```
//e- this will search for session in randomly selected world and
server ID
searchRequest.worldId= 0;
searchRequest.serverId= 0;
```

When setting the search-request parameter, the application needs to pass in the request to the `Matching::Interface::searchSession()`. If the request is successful, the function will return `SCE_TOOLKIT_NP_SUCCESS`. The application is notified by a callback event indicating that the request process is completed; the callback returned to application is `matchingSessionSearchCompleted`. To check whether the search was successfully performed, the application must check the `hasResult()` or `hasError()` of the Future object. If a search process was completed successfully, the application can get the information of the search result from the `getData()` interface function of the `Future` object.

Joining a Session

There are three ways to join a session: one uses the `Matching::Interface::joinSession()` where the application has to specify which session it wants to join, the second way uses the

`Matching::Interface::quickSession()` method, and the third method involves joining when the user accepts an invitation from the system software.

Method 1: Using Matching::Interface::joinSession

After the session information of the desired session is obtained, joining the session can be processed by sending a request to join that session (specifying the password if the session is password protected). The join request parameters are shown in Table 13.

Table 13 Join Request Parameters

Request Parameter	Structure Member	Specification
Session Information	*sessionInformation*	Required
Session Password	*sessionPassword*	Optional

The following code example shows how to set the request parameters of `Matching::Interface::joinSession`:

```
JoinSessionRequest sessionRequest;
memset(&sessionRequest,0, sizeof(JoinSessionRequest));
```

Next, specify the session information of the session that is to be joined:

```
session_to_join_info.sessionInformation = &m_sessionInformation;
```

The following code example shows how to join a session using `joinSession()`:

```
// If a session is password protected, the password must be specified
// in order to join the session.
NpMatching2SessionPassword password;

memset(&password, 0, sizeof(password));
strncpy(password.data, "ROOMPSWD", SCE_NP_MATCHING2_SESSION_PASSWORD_SIZE);
sessionRequest.sessionPassword = &password;

static sce::Toolkit::NP::Utilities::Future<
sce::Toolkit::NP::SessionInformation> joinedSessionInformation;

int ret = Matching::Interface::joinSession(&sessionRequest,
&joinsessionInformation );
if (ret != SCE_TOOLKIT_NP_SUCCESS) {
   printf("Error joining session: 0x%x",ret);
}
```

Method 2: Using Matching::Interface::quickSession

In this method the application can specify the search request by specifying the criteria for a session to search and join. If no search request is mentioned the library searches for a random session and joins the session. The quick join request parameters are shown in Table 14.

Table 14 Quick Join Request Parameters

Request Parameter	Structure Member	Specification
`SearchSessionsRequest`	`searchCriteria`	Optional

The following code example shows how to set the request parameters of `Matching::Interface::quickSession`:

```
sce::Toolkit::NP::SearchSessionsRequest searchRequest;
memset(&searchRequest , 0 , sizeof(sce::Toolkit::NP::SearchSessionsRequest));
```

Next, specify the search filter and search attributes of the session that is to be joined or alternatively you can specify NULL if you want the library to find a session and join it for you.

The following code example shows how to join a session using `quickSession()`:

```
static sce::Toolkit::NP::Utilities::Future<
sce::Toolkit::NP::SessionInformation> joinedSessionInformation;

int ret = Matching::Interface::quickSession(&searchRequest,
&joinsessionInformation );
if (ret != SCE_TOOLKIT_NP_SUCCESS) {
    printf("Error joining session: 0x%x",ret);
}
```

Method 3: Joining via Invitation

When an invite is received, a push notification is triggered. After this, `getInvitationList()` can be used to get a list of invitations and present the option of joining the sessions. Additionally, the user is able to receive an invitation via the system software when the user accepts an invite or attempts to join a session from the friends list or some other related functionality. A `SCE_SYSTEM_SERVICE_EVENT_SESSION_INVITATION` event is dispatched by the system software, which is obtainable via SceSystemService related functionality. Once the event param is received, `retrieveMessageAttachmentFromEvent()` is used to get the matching session data; `joinInvitedSession()` is then used to join the session. The following code example shows how to get a session invitation event from the system software:

```
SceSystemServiceEvent event;
sceSystemServiceReceiveEvent(&event);
switch (event.eventType)
{
case SCE_SYSTEM_SERVICE_EVENT_SESSION_INVITATION: {
    SceNpSessionInvitationEventParam* eventParam =
    (SceNpSessionInvitationEventParam)event.data.param;
    sce::Toolkit::NP::ReceiveMessageRequest rcvM;
    rcvM.eventParamData = event.data.param;
    sceUserServiceGetInitialUser(&rcvM.userInfo.userId);
    rcvM.msgType = SCE_TOOLKIT_NP_MESSAGE_TYPE_INVITE;
    sce::Toolkit::NP::Utilities::Future
       < sce::Toolkit::NP::MessageAttachment
    > sessMsgAttach;
    sce::Toolkit::NP::Messaging::Interface::
    retrieveMessageAttachmentFromEvent(&rcvM,&sessMsgAttach);
    }
break;
// events continued...
```

When the retrieval process is completed, an event is raised on the event callback and the session should then be joined. The following is an example of using the attachment obtained above to join the session:

```
sce::Toolkit::NP::Utilities::Future< sce::Toolkit::NP::SessionInformation >
    sessionInfo;
sce::Toolkit::NP::Matching::Interface::
joinInvitedSession(sessMsgAttach.get(), &sessionInfo);
```

On completion, a `matchingSessionJoined` event is raised and the `sessionInfo` Future object will contain the results.

Updating a Session

Once the session is active it is necessary for the application to call `Matching::Interface::updateSession` when an `Event::matchingSessionUpdate` or `Event::matchingSessionModified` occurs.

The above API has to be called from the NP Toolkit Thread on which the callback was triggered. If the above API is called from any other thread then an undesired behavior is expected.

The following code example demonstrates the use of the above API:

```
SceNpMatching2Event matchingEvent;
int ret = Matching::Interface::updateSession(
   &m_currentSessionInformation,&matchingEvent,event.event );

if (ret == SCE_TOOLKIT_NP_MATCHING_SESSION_KICKEDOUT ||
       ret == SCE_TOOLKIT_NP_MATCHING_SESSION_ROOM_DESTROYED) {
if (ret == SCE_TOOLKIT_NP_MATCHING_SESSION_ROOM_DESTROYED) {
       printf("Matching Room Destroyed \n");
   } else {
       printf("Kicked out of the room \n");
   }
}
```

Modifying a Session

The NP Toolkit Library also provides the functionality to modify internal or external attributes of a session. When the application needs to modify the session, it should call `Matching::Interface::modifySession()` and provide information on what type of attributes need to be modified. Please note that only one type of attribute can be modified with a single API call. The result of the request is provided through a `matchingSessionModified` event. When this event occurs, the application needs to update the session information. To do this, it should call `Matching::Interface::updateSession()` and provide it with the current `SessionInformation` structure. The application should make sure that when passing in the current `SessionInformation` structure for updating, it is not used by any other thread.

Leaving a Session

To request to leave a session, the application must call `Matching::Interface::leaveSession()`. This initiates a request to leave the currently joined session. When making the call, the application must pass in the `sessionInformation` structure of the session. If the request to kick the function off is successful, the function returns `SCE_TOOLKIT_NP_SUCCESS`. After the attempt to exit has completed, the application will be notified by a `matchingSessionLeft` callback event. The application can check whether the application was successfully able to leave or if there was an error by using the `hasResult()` or `hasError()` method of the Future object.

Even if there was an error, the application is not expected to handle the error. The `sessionInformation` structure should be cleared because the session information will no longer be valid.

Session Invitation

Sending an Invite

The NP Toolkit library provides the functionality to invite a user/other users to a currently joined session. In order to send an invite to a user, or group of users, the application needs to call the `Matching::Interface::inviteToSession()` function. This will trigger a message dialog with an attachment and a message predefined by the application to be shown. Here the user can select a friend or list of friends to invite to the current session.

Receiving an Invite

On receiving an event to join an invited session, the application should call the `Matching::Interface::joinInvitedSession()` API. This function will allow the user to join an invited session and provide the application with `SessionInformation` about the session if joining was successful. For more information regarding the error codes, please refer to the *NP Toolkit Library Reference* documentation.

List of Functions

Table 15 Matching-Related Functions

Function	Description
`registerSessionAttributes()`	Registers session attributes.
`createSession()`	Creates a session.
`searchSessions()`	Searches for a specified session.
`joinSession()`	Joins a specified session.
`modifySession()`	Modifies a specific session.
`leaveSession()`	Leaves a currently joined session.
`updateSession()`	Updates the session information of a currently joined session.
`updateSessionAttributes()`	Updates the session attributes of the current session.
`inviteToSession()`	Displays a message dialog where the user can send an invite to join the current session.
`joinInvitedSession()`	Joins a session to which the user was invited.
`quickSession()`	Searches for and joins a session.
`kickMember()`	Kicks a room member out of a room.
`registerRoomMessageCallback()`	Registers a callback which will be called for room messages.
`sendRoomMessage()`	Sends room/chat messages to the room members.

9 Messaging

The messaging service provides functionality for sending and receiving messages to other PlayStation™Network users.

Functionality

The messaging service provides functions for sending and receiving the important messages used by games on PlayStation™Network.

The service currently supports these message types:

- **Game Invites**: These messages can be sent to invite other users to join the current user in game. They have text attributes and an arbitrary data attachment defined by the developer to reference the game session to join.
- **Data Attachment Messages**: These messages are for sending arbitrary data attachments along with a text message to other users of PlayStation™Network. These messages persist in the user's inbox and when open are presented to the application. An example usage would be for sending pieces of user generated content to other users.

Sending Messages

To send any kind of message using the Messaging service the function `sce::Toolkit::NP::MessagingService::Interface::sendMessage()` is called specifying the options of the message within the parameters, for example:

```
sce::Toolkit::NP::MessageData msg;
memset(&msg,0,sizeof(msg));
msg.body = "Test your luck";
std::string datamsg = "Just another game data ";
msg.attachment = (SceChar8 *)datamsg.c_str();
msg.attachmentSize = datamsg.size();
int ret = sce::Toolkit::NP::MessagingService::Interface::sendMessage(
            &msg,SCE_TOOLKIT_NP_MESSAGE_TYPE_CUSTOM_DATA);
if (ret < 0 ) {
    printf("Fail to send message 0x%x\n",ret);
}
```

Notification of successful sending, or errors are received by the `NpToolkitCallback`. With the event type: `Event::messageSent`, or `Event::messageError`.

Displaying Received Messages

To display a list of received messages using the Messaging service the function `sce::Toolkit::NP::MessagingService::Interface::displayReceivedMessages()` should be called.

This function will display received messages using the message list dialog.

On receiving an event `SCE_APPMGR_SYSTEMEVENT_ON_NP_MESSAGE_ARRIVED`, the `sce::Toolkit::NP::MessagingService::Interface::displayReceivedMessages()` above API function can be called with `SCE_TOOLKIT_NP_MESSAGE_TYPE_INVITE` to display the game invite message.

Retrieving Message Attachment

It is possible for users to retrieve message attachment from the `SceAppUtilAppEventParam` structure returned by `sceAppUtilReceiveAppEvent()` on event `SCE_APPUTIL_APPEVENT_TYPE_NP_INVITE_MESSAGE` and `SCE_APPUTIL_APPEVENT_TYPE_NP_APP_DATA_MESSAGE` using the API call `sce::Toolkit::NP::MessagingService::Interface::retrieveMessageAttachment()`.

The message attachment can also be retrieved by calling the `sce::Toolkit::NP::MessagingService::Interface::retrieveMessageAttachmentFromId()` function. This will retrieve the attached data from the message specified by the ID.

On successful completion the application will be notified by a `NPToolkitCallback` for event `messageRetrieved` and the attached can be retrieved by accessing the `getAttachedData()` method of `MessageAttachment` class. On failure to retrieve the message attachments `sce::Toolkit::NP::Event::messageError` event is triggered

List of Functions

Table 16 Messaging Related Functions

Function	Description
`sendMessage()`	Sends a message or invite to another PlayStation™Network user.
`displayReceivedMessages()`	Displays a Received Message using the Message Dialog.
`retrieveMessageAttachment()`	Retrieves data attached to a message using `SceAppUtilAppEventParam`.
`retrieveMessageAttachmentFromId()`	Retrieves data attached to a message using `SceNpMessageId`.
`sendInGameMessage()`	Sends an in-game data message to other PlayStation™Network users.
`retrieveInGameMessage()`	Retrieves an in-game data message.

List Of Callback Events

Table 17 Messaging Related Callback Events

Event	Description
`messageRetrieved`	A message with a data attachment was received.
`messageError`	An error occurred while processing a send or retrieve request.
`messageSent`	Message sending completed.

Size Limitation

The following table shows the size restrictions for Message Data using NP Toolkit library.

Table 18 Size Restrictions for Message Data

Event	Description
Recipients	Up to 16 persons
Message body	Up to 512 bytes including terminating NULL character (UTF-8)
Attached data size	Up to 1 KB
`MESSAGE_INVITE_ACCEPTED`	Pixel size: Up to 128 x 128 pixels File size: Up to 64 KiB Format: PNG or JPEG

10 Trophy

The NP Toolkit library provides interface functions to register the trophy set, unlock and retrieve trophy information.

Note: Deprecated functions are available in addition to the new functions to maintain backwards compatibility. All of the following examples are based on the new functions.

Registering a Trophy Set

To install a trophy set, call `trophyRegisterSet()` and pass in two parameters:

- `RegisterTrophyRequest*` *`request`*
- `bool` *`async`*

The *`request`* parameter provides the information for registering a trophy set. This includes the ID of the user making the request and two boolean values. These specify whether icons and the trophy list should be cached or not. After trophy registration, information retrieved will be cached inside NP Toolkit. The trophy list will then be retrieved from the cache next time `trophyRetrieveList()` is called. Since this is an asynchronous call, success and fail events will be returned to an NP Toolkit callback.

The following code example shows the procedure for registering a trophy set.

```
RegisterTrophyRequest request;
request.cacheIcons = true;
request.cacheTrophyList = true;
int ret = Trophy::Interface::trophyRegisterSet(&request);
if (ret < 0) {
    // Error handling here
}
```

Note: This function is asynchronous and invokes a sub thread to retrieve the trophy list if `cacheTrophyList` is enabled.

Unlocking a Trophy

To install a trophy set, call `trophyUnlock()` and pass in two parameters:

- `UnlockTrophyRequest*` *`trophyRequest`*
- `bool` *`async`*

Calling this function will unlock a trophy ID specified by the *`trophyId`* member of the *`trophyRequest`* parameter. This function comes in both a synchronous and asynchronous version. A platinum trophy will be unlocked automatically when the last trophy has been unlocked. An event `Event::trophyPlatinumUnlocked` will be returned when the platinum trophy has been unlocked.

Retrieving Information About a Game's Trophy Set

To retrieve a trophy set information such as name, description, number of trophies, trophy set icon etc., call `trophyRetrieveGame()` and pass in two parameters:

- `RetrieveTrophyGameRequest *` *`request`*
- `Future<sce::Toolkit::NP::TrophyGameInfo> *` *`trophyGameInfo`*

The *`request`* parameter provides the information required to retrieve the trophy set information. This includes the ID of the user making the request. The *`trophyGameInfo`* Future object is where trophy set information such as the number of trophies, name, description and icon of the trophy set will be set to. The trophy set information cache will be automatically created once this function is called for the first time, and the `trophyGameInfo` Future object will reference this cache instead of storing its own copy in the Future object. This is why you should not try and delete the memory that *`trophyGameInfo`* is referencing. This cache memory area will be deallocated once the trophy service is terminated.

The following code example shows the procedure for retrieving trophy set information for display purposes.

```
Future<TrophyGameInfo> trophyGameInfo;
RetrieveTrophyGameRequest request;
bool async = false;
int ret = TrophyInterface::trophyRetrieveGame(&request, &trophyGameInfo,
async);
if (ret < 0) {
    printf("Trophy::Interface::trophyRetrieveGame() failed, ret 0x%x\n", ret);
    return;
}

// print out trophy set information
if (sf_trophyGameInfo.hasResult() &&
(sf_trophyGameInfo.getError() == SCE_TOOLKIT_NP_SUCCESS)) {
    printf("\n\---- Trophy Game Information ----\n");
    printf("*Title: %s\n", trophyGameInfo.get()->gameDetail.title);
    printf("*Description: %s\n", trophyGameInfo.get()
           ->gameDetail.description);
    printf("*Number of Groups: %d\n", trophyGameInfo.get()
           ->gameDetail.numGroups);
    printf("*Number of Trophies: %d\n", trophyGameInfo.get()
           ->gameDetail.numTrophies);
    printf("*Number of Platinum trophies: %d\n", trophyGameInfo.get()
           ->gameDetail.numPlatinum);
    printf("*Number of Gold trophies: %d\n", trophyGameInfo.get()
           ->gameDetail.numGold);
    printf("*Number of Silver trophies: %d\n", trophyGameInfo.get()
           ->gameDetail.numSilver);
    printf("*Number of Bronze trophies: %d\n", trophyGameInfo.get()
           ->gameDetail.numBronze);
}
```

Retrieving User Progress

To retrieve user's progress, call `trophyRetrieveProgress()` and pass in three parameters:

- `RetrieveUserTrophyProgressRequest * request`
- `Future<sce::Toolkit::NP::SceNpTrophyGameData> * gameData`
- `bool async`

The r*`equest`* parameter provides the information required to retrieve the progress of one user. This includes the ID of the user making the request. The *`gameData`* Future object is where the user's trophy progress information will be set to. Internally, every time a trophy is successfully unlocked, the user's trophy progress is stored in a cache and updated. This function simply references that cache by having the Future object point to it. This is why you should not try and delete this area of memory; instead this cache area will be deallocated when the trophy service is terminated.

The following code example shows the procedure for retrieving a user's trophy progress for displaying purposes.

```
bool isAsync = false;
Future<SceNpTrophyGameData> gameData;
RetrieveUserTrophyProgressRequest request;

int ret = Trophy::Interface::trophyRetrieveProgress(&request, &gameData,
isAsync);
```

```
    if (ret < 0) {
        printf("Trophy::Interface::trophyRetrieveProgress() failed, ret = 0x%x\n",
              ret);
    }

    if (gameData.hasResult() && (gameData.getError() == SCE_TOOLKIT_NP_SUCCESS)) {
        printf("\n---- Trophy Progress ----\n");
        printf("Progress percentage: %d\%\n", gameData.get()->progressPercentage);
        printf("Total trophies unlocked: %d\n", gameData.get()->unlockedTrophies);
        printf("Total platinum: %d\n", gameData.get()->unlockedPlatinum);
        printf("Total gold: %d\n", gameData.get()->unlockedGold);
        printf("Total silver: %d\n", gameData.get()->unlockedSilver);
        printf("Total bronze: %d\n", gameData.get()->unlockedBronze);
    }
```

Retrieving Detailed Information About a Game's Trophies

To retrieve detailed information about all the trophies in a games' trophy set, call `trophyRetrieveList()` and pass in three parameters:

- `RetrieveTrophyListRequest * listRequest`
- `Future<sce::Toolkit::NP::TrophyInfo> * trophyInfo`
- `bool async`

The `listRequest` parameter provides the information required to retrieve the information about the trophies. This includes the ID of the user making the request. The `trophyInfo` Future object is where trophy information will be set to. Note that this function will return information for every trophy in the trophy set, so make sure the `Future<sce::Toolkit::NP::TrophyInfo>` object is passed in is large enough. You can find out how many trophies there are in the trophy set by calling `trophyRetrieveGame()`. Each Future object that is passed in will reference a memory area that the trophy cache is using to store trophy information, so do not try and free this memory via the Future object. This memory area will be deallocated once the trophy service has been terminated. The trophy information cache will automatically be created once this function is called for the first time.

The following code example shows the procedure for retrieving an array of trophy information for display purposes.

```
sce::Toolkit::NP::RetrieveTrophyListRequest listRequest;
SceUserServiceUserId userId;
ret = sceUserServiceGetInitialUser(&userId);
if(ret < 0)
{
    // Error handling here
}
trophyRequest.userInfo.userId = userId;

Future<TrophyInfo> trophyInfo[numTrophies];//numTrophies can be retrieved by
trophyRetrieveGame()

ret = Trophy::Interface::trophyRetrieveList(&listRequest, &trophyInfo, false);

if(trophyInfo.hasResult() && trophyInfo.hasError() == SCE_TOOLKIT_NP_SUCCESS){
    for(int i = 0; i < numTrophies; i++) {
        // Each trophy info. in trophyInfo[i];
    }
}
```

Retrieving Trophy Group Information

To retrieve detailed information about a trophy group for a user, call `trophyRetrieveGroups()` and pass in three parameters:

- `RetrieveTrophyGroupRequest * request`
- `Future<sce::Toolkit::NP::TrophyGroupInfo> * groupInfo`
- `bool async`

The *groupInfo* Future object is where trophy group information such as number of trophies, name, description and icon of the trophy group will be set to. For the *groupId* member variable of *request*, pass in a group ID that you want to query. The trophy group information cache will be automatically created once this function is called for the first time, and the *groupInfo* Future object will reference this cache instead of storing its own copy in the Future object. This is why you should not try and delete the memory that `groupInfo` is referencing. This cache memory area will be deallocated once the trophy service is terminated.

The following code example shows the procedure for retrieving trophy group information for displaying purposes.

```
bool isAsync = false;
int ret = 0;

// Retrieve trophy set information to find out if this trophy set has
// any groups in it and how many
Future<TrophyGroupInfo> groupInfo;
RetrieveTrophyGroupRequest request;

ret = Trophy::Interface:: trophyRetrieveGroups(&request, &gameInfo, isAsync);
if (ret < 0) {
   printf("Trophy::Interface::trophyRetrieveGroups() failed, ret = 0x%x\n",
          ret);
       return;
}

int maxNumGroups = 0;
if (gameInfo.hasResult()) {
   maxNumGroups = gameInfo.get()->gameDetail.numGroups;

// Allocate memory for maxNumGroups of TrophyGroupInfo future object
// to hold them
sf_trophyGroupInfo = new Future<TrophyGroupInfo>[maxNumGroups];
if (sf_trophyGroupInfo == NULL) {
   printf("sf_trophyGroupInfo memory allocation failed");
       return;
}

// Loop based on maxNumgroups to retrieve each groups' information
isAsync = false;
for (int groupId = 0; groupId < maxNumGroups; groupId++) {
RetrieveTrophyGroupRequest request;
Request.groupId = groupId;
ret = Trophy::Interface::trophyRetrieveGroups(
      &request, &sf_trophyGroupInfo[(int)groupId], isAsync);
   if (ret < 0) {
   printf("Trophy::Interface::trophyRetrieveGroup failed, ret 0x%x \n", ret);
       }

       // Print each trophy group information here for display.
   }
}
```

Terminating the Trophy Service

To explicitly terminate a trophy service, call `trophyTerm()`. Internally this function will free internal trophy caches, unloads the trophy module and calls the following functions:

- `sceNpTrophyDestroyHandle()`
- `sceNpTrophyDestroyContext()`
- `sceNpTrophyTerm()`

Please refer to *NP Trophy Overview* for more information on these function calls.

Initializing the Trophy Service

To explicitly initialize a trophy service, call `trophyInit()`. Internally this function will load the trophy module and calls the following functions:

- `sceNpTrophyInit()`
- `sceNpTrophyCreateContext()`
- `sceNpTrophyCreateHandle()`

Please refer to the *NP Trophy Overview* document for more information on these function calls. Note that you only need to call this function to initialize the trophy service after you have called `trophyTerm()`.

List of Functions

Table 19 Trophy Related Functions

Function	Description
`trophyRegisterSet()`	Installs a trophy TRP set.
`trophyUnlock()`	Unlocks a specified trophy.
`trophyRetrieveList()`	Retrieves trophy information including `SceNpTrophyDetails`, `SceNpTrophyData`, and the icon for each trophy.
`trophyRetrieveGame()`	Retrieves a trophy set's information such as the name, description, the total number of trophies and the trophy set icon.
`trophyRetrieveProgress()`	Retrieves the user's progress for a trophy set.
`trophyRetrieveGroups()`	Retrieves trophy group information if it exists in the trophy set.
`trophyTerm()`	Explicitly terminates the trophy service.
`trophyInit()`	Explicitly initializes the trophy service.

11 Auth

Getting a Ticket

NP Toolkit provides an interface for obtaining a ticket, which corresponds to a service ID on the PlayStation™Network server. This ticket can be used to authenticate PlayStation™Network users on external servers when used in conjunction with the Ticket Checker Module (NP TCM).

To use this functionality you must register your Service ID first with NP Toolkit using `sce::Toolkit::NP::Interface::registerServiceId()`.

The following code example shows the procedure for obtaining a ticket.

```
int ret;
Future<Ticket> ticket;

ret = Auth::Interface::getTicket(&ticket);

if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

Note: `getTicket()` is an asynchronous function. The ticket Future variable must remain in scope until the `authGotTicket` or `authError` event has been received in the NP Toolkit callback.

List of Functions

Table 20 Ticket Related Functions

Function	Description
`getTicket()`	Gets a ticket from the PlayStation™Network server.
`getCachedTicket()`	Retrieves a cached ticket if one is available and valid.

12 Commerce

NP Toolkit offers a simplified API for that provides browsing functions and enables products to be purchased from the PlayStation®Store from in-game. To use this functionality you must register your Service ID first with NP Toolkit using `sce::Toolkit::NP::Core::registerServiceId()`.

This document does not provide information on creating and preparing the products for the title store. Detailed information of PlayStation®Store, and the specifications of products distributed through PlayStation®Store can be found in the following documents:

- *PSN ^{SM} Commerce Service Overview*
- *PSN ^{SM} Commerce Programming Guide*
- *PlayStation®Store Content Guidelines for PlayStation®Vita*

Information of making products available in PlayStation®Store can be found in the following document:

- *NP Product Management Guide*

Information for handling DRM additional content can be found in the following document:

- *Application Utility Overview*

Creating a Commerce Session

Before using the in-game browsing functions `getCategoryInfo()`, `getProductList()` or `getDetailedProductInfo()`, the application must first create a commerce session. The created commerce session will subsequently be used in obtaining category content data and product data.

To create a commerce session simply call `createSession()`. The function is asynchronous only will return immediately. When a session has been created successfully, the event `commerceSessionCreated` will be sent to the NP Toolkit callback. Example usage:

```
int ret;
ret = Commerce::Interface::createSession();
if (ret < 0) {
    // Error handling
}

// Wait for commerceSessionCreated event
```

Note: A created commerce 2 session will be deleted on the server-side when a specific amount of inactivity time passes. When the session is deleted from the server-side, the `SCE_NP_COMMERCE2_SERVER_ERROR_SESSION_EXPIRED` error will return for any subsequent communication attempt. To resume communication from this state, call `createSession()` again to create a new session.

Currently, the session time-out time set on the server–side is 15 minutes, however, this value may be changed in the future. Do not program your application in such a way that it is dependent on this time. In addition, server load may be negatively affected when communication is periodically generated to prevent a session time-out; please do not implement this type of processing in your application.

Getting Store Category Information

Information about the categories that have been set up on the PlayStation®Store can be obtained by calling `getCategoryInfo()`. This information includes the category descriptions, category ID, image URL, number of products in that category and information of any the immediate sub-categories contained within it. You can specify a certain category ID to the second paramemeter of `getCategoryInfo()` to obtain information from that category, or else you can specify `NULL` to obtain information about the root category.

Before obtaining this information you will need to create a commerce session. The following code example shows the procedure for obtaining category information:

```
int ret;
Future<CategoryInfo> info;

ret = Commerce::Interface::getCategoryInfo(&info, NULL, false);
if (ret < 0) {
    // Error handling
}
```

Getting a List of Products on the Store

A list of products that are available to a user on the PlayStation®Store can be easily retrieved by calling `getProductList()`. This information includes the product ID, product name, product description, product image, price, purchasabiltiy flag and release date. You can specify a certain category ID to the second paramemeter of `getProductList()` to retrieve a list of products from that category, or else you can specify `NULL` to retrieve products from the root category.

Before obtaining this information you will need to create a commerce session. The following code example shows the procedure for obtaining a list of products:

```
int ret;
Future<std::vector<ProductInfo> > productList;

ret = Commerce::Interface::getProductList(&productList, NULL, false);
if (ret < 0) {
    // Error handling
}
```

Note: To obtain basic SKU information such as price and release date for each item returned, a one to one mapping of SKUs to products is needed. So, to ensure that these members are filled, make sure to only create one SKU per product in the Network Product Management Tool (NPMT).

Getting Detailed Product Information

When rendering your own custom storefront in-game, you must use this function to remain TRC compliant. Use `getDetailedProductInfo()` to obtain and show all relevant product information to the user before allowing them to proceed directly to the checkout.

To obtain detailed information about a product, call `getDetailedProductInfo()` specifying the product ID and the category ID that it belongs to. Detailed product information contains basic product information plus the product long description, legal description, SKU and rating information. Before obtaining this information you will need to create a commerce session.

The following code example shows the procedure for obtaining detailed product information:

```
int ret;
Future<ProductInfoDetailed> productInfo;

// Specify product ID
ret = Commerce::Interface::getDetailedProductInfo(&productInfo,
"EP0000-NPXX12345_00-0123456789ABCDEF", NULL, false);
if (ret < 0) {
    // Error handling
}
```

Note: Instead of hard coding product IDs, you can retrieve a list of products programmatically using `getProductList()`. As with items returned with `getProductList()`, this function assumes a one to one mapping of SKUs to products in the Network Product Management Tool (NPMT), for simplicity. Use this function in conjunction with `Commerce::Interface::checkout()`.

Store Category Browsing

With Store category browsing, the application is suspended, and all the processing necessary for browsing and purchasing is handled by the system software. The category to start the browsing can be specified by the application. For browsing to start at the root category, `NULL` can be specified instead of a category ID. After all processing completes, the application process is automatically restarted.

The following code example shows the procedure to start Store category browsing:

```
int ret;

ret = Commerce::Interface::categoryBrowse(NULL);
if (ret < 0) {
    // Error handling
}
```

Store Product Browsing

With Store product browsing, the application is suspended, and all the processing necessary for browsing and purchasing is handled by the system software. The product to launch the Store to is specified by the application. After all processing completes, the application process is automatically restarted.

The following code example shows the procedure to start Store product browsing:

```
sce::Toolkit::NP::ProductBrowseParams params;
strcpy(params.productId, "ED1987-NPXX00877_00-ENT0010000000000");

int ret = Commerce::Interface::productBrowse(params, false);
if (ret < 0) {
    // Error handling
}
```

Promotion Code Redemption

When using promotion code redemption, the application is suspended, and all the processing necessary for redeeming a promotion code is handled by the system software. After all processing completes, the application process is automatically restarted.

The following code example shows the procedure to start promotion code redemption:

```
ret = Commerce::Interface::voucherCodeInput(false);
if (ret < 0) {
    // Error handling
}
```

Checkout

If your title renders its own custom in-game store, assuming it is TRC compliant, it can bring the user straight to the checkout, where the user can purchase a number of SKUs.

When the `checkout()` function is called, a system checkout overlay is displayed to the user in the foreground while your application continues to run in the background. The `commerceCheckoutStarted` event will be sent to the NP Toolkit callback when the overlay is initially displayed.

The following code example shows the procedure to launch the checkout:

```
std::list<char *> skuList;

// productInfo was retrieved earlier using getDetailedProductInfo()
if (productInfo.hasResult()) {
    skuList.push_back(productInfo.get()->skuId);
}

int ret = Commerce::Interface::checkout(skuList, false);
if (ret < 0) {
    // Error handling
}
```

When the user has ended the checkout process by either canceling or purchasing, the overlay will be removed and the `commerceCheckoutFinished` event will be sent to the NP Toolkit callback. Depending on the users operation, the following will be returned as the events return code:

Table 21 Checkout Event Return Codes

Value	Hexadecimal	Description
SCE_COMMON_DIALOG_RESULT_OK	0x0	Processing complete.
SCE_COMMON_DIALOG_RESULT_USER_CANCELED	0x1	User performed cancel operation.
SCE_STORE_CHECKOUT_DIALOG_ERROR_INTERNAL	0x80102201	Internal error.
SCE_STORE_CHECKOUT_DIALOG_ERROR_PARAM	0x80102202	Parameter error.

Note: `sceCommonDialogUpdate()` must be called every frame until the `commerceCheckoutFinished` event has been sent to the NP Toolkit callback.

Download List

To add functionality to re-download content in-game, your title can make use of the `displayDownloadList()` function. This displays a list of DRM content to the user that they have already purchased for your title. You can specify specific SKUs to display to the user in the *`skuList`* parameter. Otherwise, to display all available content, pass in an empty SKU list.

When the `displayDownloadList()` function is called, a system overlay is displayed to the user in the foreground while your application continues to run in the background. The `commerceDownloadListStarted` event will be sent to the NP Toolkit callback when the overlay is initially displayed.

The following code example shows the procedure to launch the download list:

```
std::list<char *> skuList;

int ret = Commerce::Interface::displayDownloadList(skuList, false);
if (ret < 0) {
    // Error handling
}
```

When the user has ended the download list process by either canceling or downloading, the overlay will be removed and the `commerceDownloadListFinished` event will be sent to the NP Toolkit callback.

Note: `sceCommonDialogUpdate()` must be called every frame until the `commerceDownloadListFinished` event has been sent to the NP Toolkit callback.

Getting a List of Service Entitlements

Call `getEntitlementList()` to get a list of Service Entitlements associated with the current PlayStation™Network user. For the argument, `list`, specify a Future variable for receiving the obtained result.

The following code example shows the procedure for retrieving a list of service entitlements:

```
int ret;
Future<std::vector<SceNpEntitlement> > list;

ret = Commerce::Interface::getEntitlementList(&list);
if (ret < 0) {
    // Error handling
}

// The commerceGotEntitlementList event will be sent to the
// NP Toolkit callback when processing has completed.
```

Note: `getEntitlementList()` is an asynchronous only function.

Consuming a Service Entitlement

Call `consumeEntitlement()` to decrement uses of a consumable entitlement by a specified amount. For the argument, *`id`*, specify the entitlement ID of the consumable entitlement and for the argument, *`consumedCount`*, specify the number of uses that you wish to consume. This function uses the `auth` library internally, so when an entitlement has bee consumed successfully the `auth gotTicket` event will be sent to the NP Toolkit callback. Conversly, the *`authError`* event will be sent to the NP Toolkit callback if an error has occurred.

The following code example shows the procedure for consuming an entitlement:

```
int ret;

ret = Commerce::Interface::consumeEntitlement(id, consumedCount);

if (ret < 0) {
    // Error handling
}
```

Background Download Status

In most cases when additional content is purchased within a game, it is added to the background download list. Content that has downloaded in the background while the game is running is not normally installed and mounted until the application has restarted. Call `getBgdlStatus()` to get information on additional content that is either currently downloading in the background or has completed downloading.

The following code example shows the procedure for retrieving the background download status:

```
Future<SceAppUtilBgdlStatus> status;

int ret = Commerce::Interface::getBgdlStatus(&status, false);
if (ret < 0) {
    // Error handling
}
```

Installing Additional Content

Content that is downloaded in the background is not normally installed and mounted until the application has been restarted. However, by calling the `installContent()` function, the application is able to install and mount additional content and full game upgrades in the foreground without restarting the application.

The following code example shows the procedure for installing content in the foreground:

```
int ret = Commerce::Interface::installContent(false);
if (ret < 0) {
    // Error handling
}
```

Note: Verify that there is content ready to be installed by checking the background download status using `getBgdlStatus()`. If `installContent()` is called when there are no items are install, an error message will be presented to the user.

List of Functions

Table 22 Commerce Related Functions

Function	Description
`createSession()`	Creates a commerce session.
`getCategoryInfo()`	Gets information about a specified Store category.
`getProductList()`	Gets a list of products from a specified category.
`getDetailedProductInfo()`	Gets all available information about a specified product.
`categoryBrowse()`	Launches the Store to a specified category.
`productBrowse()`	Launches the Store to a specified product.
`voucherCodeInput()`	Launches the Store so the user can redeem a promotion code.
`getEntitlementList()`	Gets the list of service entitlements associated with the user.
`consumeEntitlement()`	Consumes a specified amount from a consumable entitlement.
`checkout()`	Displays the checkout overlay where the user can confirm purchase of specified SKUs.
`displayDownloadList()`	Displays a download list to the user where they can redownload content.
`getBgdlStatus()`	Gets the status of additional content that was downloaded or is being downloaded in the background.
`installContent()`	Installs and mounts all content that was downloaded in the background.

13 SNS

NP Toolkit provides an interface for posting a message to a user's Facebook "Wall". The message can contain user-specified text and a photo with an accompanying title, caption and description. It is also possible to specify a Facebook "action link".

Posting a Message to Facebook

Firstly set the Facebook application ID by calling `setAppIdFb()` and passing in one parameter:

- `uint64_t id`

Then post the message by calling `postMessageFb()` and passing in one parameter:

- `sce::Toolkit::NP::MessageDetails msgDetails`

Use the `setAppIdFb()` function to set the Facebook Application ID, which is stored within the service. This ID is supplied by Facebook when an application is created. The ID must be set before attempting to post a message; otherwise all attempts to retrieve an access token will fail. The `postMessageFb()` function expects a `MessageDetails` object to be passed into the `msgDetails` parameter. This holds all the information necessary to describe the contents of a message that will be posted to Facebook. The `MessageDetails` structure consists of a string to store user-text, a sub-structure (`PhotoFb`) to store photo details and another sub-structure (`ActionLinkFb`) to describe a Facebook "actionLink". The `PhotoFb` member stores four strings: the URL of the image, the title, caption and description. The `ActionLinkFb` member stores two strings: the URL of the link and the custom text to be displayed.

If `postMessageFb()` is called and the user has not bound their Facebook account to the PlayStation®Vita system, they will be presented with a dialog in which to enter their Facebook login credentials. When the dialog begins, an `snsDialogStarted` event will be sent to the NP Toolkit event handler. When the user has finished with the dialogue, an `snsDialogFinished` event will be sent to the NP Toolkit event handler.

Internally, the SNS service will automatically request an access token from Facebook with the "`publish_stream`" permission. If the user has not already granted this permission to your application, they will be presented with a message dialogue to either accept or decline. When the dialog begins, an `snsDialogStarted` event will be sent to the NP Toolkit event handler. When the user has finished with the dialogue, an `snsDialogFinished` event will be sent to the NP Toolkit event handler.

If a completely blank `MessageDetails` object is passed to the `msgDetails` parameter, an error will be returned. The `userText`, `photoFb` and `actionLink` members can be left blank as long as at least one of them contains data. The following code example shows the procedure to set the application ID and then post a message to a user's Facebook Wall:

```
int ret;

// Set Facebook App ID
    uint64_t id = 139345646095149ULL;

ret = sce::Toolkit::NP::Sns::Interface::setAppIdFb(id);
if(ret < 0){
    // Error handling
}

    sce::Toolkit::NP::MessageDetails msgDetails;

    msgDetails.userText = "Test";

msgDetails.photo.url = "http://...";
msgDetails.photo.title = "Title";
msgDetails.photo.caption = "This is caption";
```

```
    msgDetails.photo.description = "This is a description";

    msgDetails.actionLink.name = "Go To PlayStation.com";
    msgDetails.actionLink.url = "http://uk.playstation.com/";

    ret = sce::Toolkit::NP::Sns::Interface::postMessageFb(msgDetails);

    if(ret < 0){
        // Error handling
    }
```

If a message has sucessfully been posted to a user's Facebook wall, a `snsMessagePosted` event will be sent to the NP Toolkit event handler.

Note: The `postMessageFb()` function only operates asynchronously as it can take some time to communicate with the Facebook Graph API. The `setAppIdFb()` function only operates synchronously.

List of Functions

Table 23 SNS Related Functions

Function	Description
`setAppIdFb()`	Sets the Facebook application ID.
`postMessageFb()`	Posts a message to the user's Facebook wall

14 “near”

NP Toolkit provides an interface for creating and registering a gift ready to be distributed. In addition, retrieval functionality is provided for gift information, gift images, gift data, nearby users, and “near” status.

Initializing the “near” service

Unlike other services where initialization is done implicitly, it is the application’s responsibility to initialize the “near” service. This is achieved by passing in a version and the size of the work memory to be used by the library. Work memory is allocated and managed internally by NP Toolkit and is not exposed on the application side. When specifying the size of the work memory, pass in a size of `SCE_NEAR_UTIL_DEFAULT_WORKMEMORY_SIZE` or larger. If a size below `SCE_NEAR_UTIL_DEFAULT_WORKMEMORY_SIZE` is passed, NP Toolkit will use `SCE_NEAR_UTIL_DEFAULT_WORKMEMORY_SIZE` as the work memory size.

On top of the allocation for work memory to be used by the “near” utility library, a work memory area of `SCE_NEAR_GIFT_IMAGE_SIZE` for images, and a work memory area of `SCE_NEAR_GIFT_DATA_MAX_SIZE` for gift data will also be allocated by NP Toolkit. Image and data retrieval processes will store image and body data in these areas respectively.

Retrieving Nearby Users

Nearby users are users who the current user has met at locations visited via the “near” system application. Using a single interface, it is possible to retrieve 3 types of nearby user:

- `sceNpToolkitNPNeighborDefault`
- `sceNpToolkitNPNeighborRecent`
- `sceNpToolkitNPNeighborNew`

The `sceNpToolkitNPNeighborDefault` type will return up to 100 nearby users that the current user has met including both old and new. The `sceNpToolkitNPNeighborRecent` type will return only recent nearby users met since the last “Update” activity was performed by the “near”system application. Finally, the `sceNpToolkitNPNeighborNew` type will return only new users discovered since the last “Update” activity performed by the “near” system application.

Example:

```
int ret = 0;
Future<NearNeighbors> neighbors;

ret = Near::Interface::getNeighbor(&neighbors,
sce::Toolkit::NP::sceNpToolkitNPNeighborDefault, false);
if (ret < 0) {
    printf("Near::Interface::getNeighbor() failed, ret = 0x%x\n", ret);
    return;
}

printf("\n**Neighbor retrieved**\n");
for (int i = 0; i < neighbors.get()->arraySize; i++) {
    printf("%s\n", neighbors.get()->neighbors[i].handle.data);
}
```

Retrieving the Current User's "near" Status

The "near" system application keeps track of current user's "near" activities such as the distance travelled, the number of gifts discovered, the number of nearby users discovered, and number of titles discovered. This status information can be retrieved and used in the game application.

Example:

```
Future<SceNearMyStatus> myStatus;
int ret = Near::Interface::getMyStatus(&myStatus, false);
if (ret < 0) {
    printf("Near::Interface::getMyStatus() failed, ret = 0x%x\n",ret);
    return;
}

if (myStatus.hasResult() && myStatus.hasError() == SCE_TOOLKIT_NP_SUCCESS)
{
    printf("\n**User's near Status**\n");
    printf("Number of items discovered: %llu\n",
            myStatus.get()->discoveredItemNum);
    printf("Number of title discovered: %llu\n",
            myStatus.get()->discoveredTitleNum);
    printf("Number of people encountered: %llu\n",
            myStatus.get()->encounterNum);
    printf("Distance travelled: %f\n", myStatus.get()->travelDistance);
}
```

Creating and Registering a "near" Gift for Distribution

To create and register a gift for distribution, call `Near::Interface::CreateGift()`. A `sce::Toolkit::NP::NearGiftInputParam` object containing the appropriate values should be passed in. Once the gift is created, NP Toolkit will also internally register the gift for distribution.

Note that the *`giftId`* that is passed in during gift creation is only used by the "near" system application. It is not possible to later reference this gift ID on the receiving side. Instead, the "near" utility references the discovered gifts by using `SceNearGiftDiscoveringId`. This is just an index into an array that contains the discovered gifts. Because of this, NP Toolkit embeds the *`giftId`* inside the gift's header section and thereby enables a discovered gift to be later referenced by its gift ID.

A gift's header section also contains the magic number, gift type, the number of users who have received this gift and the 10 `SceNpIds` of users who has received this gift before relaying it on. In total 256 bytes are reserved by the gift's header leaving `SCE_TOOLKIT_NP_MAX_GIFT_BODY_SIZE` for the gift content itself. Because of the use of header sections for the gift, gifts created using NP Toolkit are not compatible with gifts outside of NP Toolkit. The magic number is there to gate this behaviour, and it acts like an ID that specifies if the gift was created using NP Toolkit or not.

`Near::Interface::createGift()` is not thread safe. This is because gift content, image file paths and data file paths that are passed in are loaded into the memory areas, which are created internally during the `initNear()` call. These memory areas can only hold one gift image and one piece of gift data at a time only.

Example:

```
NearGiftInputParam giftDetails;
memset(&giftDetails, 0x00, sizeof(giftDetails));

giftDetails.name = "NP Toolkit gift name";
giftDetails.description = "NP Toolkit gift description";
giftDetails.iconPath = SAMPLE_A_IMAGE_FILENAME;
giftDetails.dataPath = SAMPLE_A_DATA_FILENAME;
giftDetails.giftId = 0x00000008;
giftDetails.giftUnits = 10;
giftDetails.giftCondition.radius = 50;
giftDetails.giftCondition.duration = 3* 24;
giftDetails.giftCondition.probability = 100;
giftDetails.giftCondition.receiverAttrs.playerRelation =
    SCE_NEAR_PLAYER_RELATION_FRIEND | SCE_NEAR_PLAYER_RELATION_PLAYER;

int ret = Near::Interface::createGift(&giftDetails, false);
if (ret < 0) {
    printf("Near::Interface::createGift() failed, ret = 0x%x\n", ret);
}
```

Retrieving Gift Information

Before calling `Near::Interface::getGiftDetails()` to retrieve the gift, the application is required to call `sceNearGetDiscoveredGifts()` to find out how many discovered gifts are available. Gift information includes the following details:

- Gift name
- Gift description
- Gift Status
- Gift Sender

Example:

```
// First retrieve number of discovered gifts by calling
// sceNearGetDiscoveredGifts()

Future<sce::Toolkit::NP::NearDiscoveredGiftDetails> giftInfo;
for (int i = 0; i < numGifts; i++) {
    int ret = Near::Interface::getGiftDetails(giftInfo, s_discoveredIdArray[i],
                                            false);

    if (ret < 0) {
        printf("getGiftDetails() failed, ret = 0x%x\n", ret);
        return;
    }

    printf("Gift name: %s\n", giftInfo->get()->giftInfo.giftName);
    printf("Gift description: %s\n",
           giftInfo->get()->giftInfo.giftDescription);
    switch (giftInfo->get()->giftStatus) {
    case 0:
        printf("Gift status: Discovered\n");
        break;
    case 1:
        printf("Gift status: Received\n");
        break;
    case 2:
        printf("Gift status: Expired\n");
        break;
    default:
        printf("Gift status: unknown\n");
    }
    printf("Gift sender: %s\n", giftInfo->get()->giftSender.handle.data);
    s_giftInfo->reset();
}
```

Retrieving Gift Image Data

Before calling `Near::Interface::getGiftImage()` to retrieve the gift image data, the application is required to call `sceNearGetDiscoveredGifts()` to find out how many discovered gifts are available and their `SceNearGiftDiscoveringId`.

A pointer to the image work memory area containing the gift image is returned on success. Because every retrieved gift image is placed into this same memory, the application has to copy image data from this work memory area to the local memory area before retrieving another gift image. This avoids overwrite.

Example:

```
// First retrieve number of discovered gifts by calling
// sceNearGetDiscoveredGifts()

Future<sce::Toolkit::NP::NearDiscoveredGiftImage> giftImage;
int ret = Near::Interface::getGiftImage(giftImage, s_discoveredIdArray[0],
false);
if (ret < 0) {
    printf("getGiftImage() failed, ret = 0x%x\n", ret);
} else {
    printf("gift image retrieved\n");
}
```

Retrieving Gift Body Data

Before calling `Near::Interface::getGiftImage()` to retrieve the gift body data, the application is required to call `sceNearGetDiscoveredGifts()` to find out how many discovered gifts are available and their `SceNearGiftDiscoveringId`. Note that only the "Received" gift's body data can be retrieved. Gifts other than "Received" will return an error.

A pointer to data work memory area containing the gift data is returned on success. Because every retrieved gift body data is placed into this same work memory, the application has to copy body data from this work memory area to the local memory area before retrieving another piece of gift data. This avoids overwrite.

Example:

```
// First retrieve number of discovered gifts by calling
// sceNearGetDiscoveredGifts()

Future<sce::Toolkit::NP::NearDiscoveredGiftData> giftData;

int ret = Near::Interface::getGiftData(giftData, s_discoveredIdArray[0],
false);
if (ret < 0) {
    printf("getGiftData() failed, ret = 0x%x\n", ret);
} else {
    printf("gift data retrieved\n");
    printf("Gift data header:\n");
    printf("Gift ID: %d\n", giftData->get()->pGiftData->header.giftId);
    printf("Gift Type: %d\n", giftData->get()->pGiftData->header.giftType);
    printf("Number of users: %d\n",
           giftData->get()->pGiftData->header.numberOfUsers);
    for (int i = 0; i < giftData->get()->pGiftData->
         header.numberOfUsers; i++) {
        printf("%s, ", giftData->get()->pGiftData->header.npId[i]);
    }
    printf("\n");
}
```

Comparing Gift IDs

To check whether a gift referenced by an `SceNearGiftDiscoveringId` (returned from calling `sceNearGetDiscoveredGifts()`) has a specified gift ID embedded inside the its header , call `Near::Interface::compareGiftId()`. The `SceNearGiftDiscoveringId` of the gift that you want to check and gift ID to be checked against should be passed in. `SCE_TOOLKIT_NP_SUCCESS` will be returned if a match is found, and `SCE_TOOLKIT_NP_GIFT_NOT_MATCH` will be returned if the specified gift ID is not embedded into the gift specified by `SceNearGiftDiscoveringId`.

Example:

```
// First retrieve number of discovered gifts by calling
// sceNearGetDiscoveredGifts()

int ret = Near::Interface::compareGiftId(8, s_discoveredIdArray[0]);
if (ret == 0) {
    printf("Gift matches\n");
} else {
    printf("Gift ID does not match\n");
}
```

Rewrapping “Received” Gifts and Registering Them for Distribution

Once a gift has been received and opened, it is possible to wrap the gift and register it for distribution again. Internally, the current user’s `SceNpId` is embedded into the header of the gift. This enables users who receive this gift to know who else has received this gift and passed it on. To do this, call `Near::Interface::relayGift()` passing in `sce::Toolkit::NP::NearRelayGiftParam` with appropriate values.

Example:

```
// First retrieve number of discovered gifts by calling
sceNearGetDiscoveredGifts()

NearRelayGiftParam giftParam;
memset(&giftParam, 0x00, sizeof(NearRelayGiftParam));
giftParam.discoveringGiftId = 1; // First gift
giftParam.giftUnits = 10;
giftParam.giftCondition.radius = 50;
giftParam.giftCondition.duration = 3* 24; // Days
giftParam.giftCondition.probability = 100;
giftParam.giftCondition.receiverAttrs.playerRelation =
SCE_NEAR_PLAYER_RELATION_FRIEND | SCE_NEAR_PLAYER_RELATION_PLAYER;
int ret = Near::Interface::relayGift(&giftParam, false);
if (ret < 0) {
    printf("nearRelayGift() failed, ret = 0x%x\n",ret);
} else {
    printf("Gift relayed successfully");
}
```

Launching the “near” System Application

To launch the “near” service, call `Near::Interface::launchNearApp()` and pass in the type of launch and the `SceNearGiftDiscoveringId`.

There are two types of launch:

- `SCE_NEAR_APP_ACTION_UPDATE`
- `SCE_NEAR_APP_ACTION_TAKE_GIFT`

`SCE_NEAR_APP_ACTION_UPDATE` will suspend the application and launch the "near" system application. Once the "near" system application is launched, the user will be presented with a message advising them to carry out an "Update" operation in order to sync with the "near" server.

`SCE_NEAR_APP_ACTION_TAKE_GIFT` will suspend the application and launch the "near" system application. Once the "near" system application is launched, the user is taken to a gift, specified by a `SceNearGiftDiscoveringId`, so they can download it.

If the `SCE_NEAR_APP_ACTION_TAKE_GIFT` launch type is chosen, the `SceNearGiftDiscoveringId` should be passed as the *`giftID`* parameter; otherwise 0 should be passed in.

Example:

```
int ret = Near::Interface::launchNearApp(SCE_NEAR_APP_ACTION_UPDATE, 0);
if (ret < 0) {
    printf("launchNearApp() failed, ret = 0x%x\n", ret);
}
```

Terminate "near" Service

The "near" service can be terminated when not in used without having to terminate the whole NP Toolkit. It is recommended to terminate "near" service to save memory when it is not in use. For example, if the current user is outside of the "near" section of the game and the "near" service has its own in-game menu, the service should be terminated.

List of Functions

Table 24 "near" Related Functions

Function	Description
`initNear()`	Initializes the "near" service taking in the version and size of the work memory for the "near" utility.
`termNear()`	Terminates the "near" service and frees the memory allocated when calling `initNear()`.
`getNeighbor()`	Retrieves the current user's nearby users.
`getMyStatus()`	Retrieves the current user's "near" status.
`createGift()`	Creates and then registers a "near" gift to be distributed.
`getGiftDetails()`	Retrieves information about "Discovered"/"Received" gifts such as the status, sender, name and description.
`getGiftImage()`	Retrieves a "Discovered"/"Received" gift's image size and data.
`getGiftData()`	Retrieves a "Received" gift's body data.
`compareGiftId()`	Compares a specified gift ID against an embedded gift ID to see if they match.
`relayGift()`	Rewraps "Received" gifts and registers them for distribution.
`launchNearApp()`	Launches the "near" system application.

15 Word Filter

NP Toolkit provides an interface with which to either censor or sanitize comments (or singular words).

Censoring a Comment

Call the `filterWorld()` function and pass in NULL for the *`sanitizedComment`* parameter. Use a `WordFilterParam` object to pass in the comment to censor and also to specify whether to trigger an asynchronous or synchronous function. The function returns a value of `SCE_NP_COMMUNITY_SERVER_ERROR_CENSORED` if inappropriate words are present in the content.

The following code example shows the procedure to censor a comment:

```
sce::Toolkit::NP::WordFilterParam censorComment;
memset(&censorComment, 0x00, sizeof(censorComment));
censorComment.isAsync = false;
memcpy(censorComment.comment, g_originalComment, sizeof(g_originalComment));

int ret = sce::Toolkit::NP::WordFilter::Interface::filterWord(NULL,
&censorComment);
if (ret < 0) {
   // Error handling
   if (SCE_NP_COMMUNITY_SERVER_ERROR_CENSORED) {
      // Comment contains inappropriate words
   }
}
```

Sanitizing a Comment

Call the `filterWorld()` function and pass in a Future object of the `WordFilterSanitized` type to the *`sanitizedComment`* parameter. Use a `WordFilterParam` object to pass in the comment to sanitize and also to specify whether to trigger an asynchronous or synchronous function. In the sanitized version of the comment returned via the Future object, inappropriate words are replaced with a `'*'`'s of the same length as the original word.

The following code example shows the procedure to sanitize a comment:

```
sce::Toolkit::NP::WordFilterParam censorComment;
memset(&censorComment, 0x00, sizeof(censorComment));
censorComment.isAsync = false;
memcpy(censorComment.comment, g_originalComment, sizeof(g_originalComment));
sce::Toolkit::NP::Utilities::Future<sce::Toolkit::NP::WordFilterSanitized>
wordFilterSanitized;

int ret = sce::Toolkit::NP::WordFilter::Interface::filterWord(
&wordFilterSanitized, &censorComment);
if (ret < 0) {
   // Error handling
}

if (wordFilterSanitized.hasResult() && wordFilterSanitized.get()->size > 0) {
   printf("Sanitized comment: %s\n",
wordFilterSanitized.get()->sanitizedComment);
}
```

List of Functions

Table 25 Word Filter Related Functions

Function	Description
`filterWord()`	Censors or sanitizes comments (or singular words).

16 Title User Storage (TUS)

The NP title user storage (TUS) utility is a system utility for using the storage that is provided per title and per user on the PlayStation™Network server.

When using the title user storage service the following data can be stored on the PlayStation™Network server:

- 32 64-bit integers (TUS variables) per user.
- Binary data (TUS data) of up to 1MB per user (PlayStation™Network account).
- 128 64-bit integers (TUS variables) per virtual user.
- Binary data (TUS data) of up to 2MB per virtual user.

Settings can be made for each piece of data so that read/write operations by other users onto that area can be either enabled or disabled. A maximum of 8 virtual users are allowed per title as described in the *NP TUS Library Overview* document.

In addition to simple read and write operations, atomic additions and conditional writes can be performed on a TUS variable. For information regarding TUS data and variables please refer to the *NP TUS Library Overview* document. It is also possible to operate on multiple TUS variables belonging to a single user at the same time, and to operate on one particular TUS variable common to multiple users at the same time.

The design of the title user storage is such that updates and references are made on the same database. Because of this there is no delay in gathering and analyzing information.

Before using this service for your title, you must set up the slot and data attributes on the SMT.

Setting TUS Variables

Call `setVariables()` to set the TUS variables for a given user. For the *`params`* argument, use the input parameters to specify the variables to update and the values to update them with. To reduce calls to the TUS server, multiple TUS variables can be set in one call.

The following code example shows the procedure for setting TUS data for a given user.

```
int ret;
Future<SceNpId> npid;
TusSetVarsInputParams params;

ret = UserProfile::Interface::getNpId(&npid, false);
if (ret < 0) {
    // Error handling
}

// Add some test variables for the current user
params.npid = *npid.get();
params.vars.push_back(TusVariable(1, 12345));
params.vars.push_back(TusVariable(2, -1));
params.vars.push_back(TusVariable(3, 1337));
params.isVirtualUser = false;

ret = TUS::Interface::setVariables(params, false);
if (ret < 0) {
    // Error handling
}
```

If `setVariables()` is called asynchronously, a `tusVariablesSet` event will be sent to the NP Toolkit callback on successful completion.

Note: To perform this operation on a virtual user, assign true to `params.isVirtualUser` and add the virtual user's handle to `params.npid.handle`.

Getting TUS Variables

Call `getVariables()` to obtain TUS variables for a given user. For the *`vars`* argument, specify a Future object for receiving the variables, and for the *`params`* argument, use the input parameters to specify the variables to retrieve. Getting multiple TUS variables at one time reduces calls to the TUS server.

The following code example shows the procedure for obtaining TUS variables for a given user.

```
int ret;
Future<SceNpId> npid;
Future<std::vector<SceNpTusVariable> > vars;
TusGetVarsInputParams params;
int32_t slots[3] = {1, 2, 3}; // Just some test slots

ret = UserProfile::Interface::getNpId(&npid, false);
if (ret < 0) {
    // Error handling
}

params.npid = *npid.get();
params.slotIds = slots;
params.numSlots = 3;
params.isVirtualUser = false;

ret = TUS::Interface::getVariables(&vars, params, false);
if (ret < 0) {
    // Error handling
}
```

If `getVariables()` is called asynchronously, a `tusVariablesReceived` event will be sent to the NP Toolkit callback on successful completion.

Note: To perform this operation on a virtual user, assign true to `params.isVirtualUser` and add the virtual user's handle to `params.npid.handle`.

Setting TUS Data

Call `setdata()` to set the TUS data for a given user. For the *`params`* argument, use the input parameters to specify the updated data and the slot it is for. TUS data can only be set for one slot at a time.

The following code example shows the procedure for setting TUS data for a given user.

```
int ret;
Future<SceNpId> npid;
TusSetDataInputParams params;
char * data = "This is some arbitrary data";

ret = UserProfile::Interface::getNpId(&npid, false);
if (ret < 0) {
    // Error handling
}

// Add some test data for the current user
params.slotId = 1;
params.npid = *npid.get();
params.data.buffer = (void *)data;
params.data.bufferSize = strlen(data);
params.isVirtualUser = false;

ret = TUS::Interface::setData(params, false);
if (ret < 0) {
    // Error handling
}
```

If `setData()` is called asynchronously, a `tusDataSet` event will be sent to the NP Toolkit callback on successful completion.

Note: To perform this operation on a virtual user, assign true to `params.isVirtualUser` and add the virtual user's handle to `params.npid.handle`.

Getting TUS Data

Call `getdata()` to obtain TUS data for a given user. For the *`data`* argument, specify a Future object of `TusDataOutput` type to receive the data in, and for the *`params`* argument, use the input parameters to specify the slot to obtain the data for. TUS data can only be retrieved from one slot at time.

The following code example shows the procedure for obtaining TUS data for a given user.

```
int ret;
Future<SceNpId> npid;
Future<TusDataOutput> data;
TusGetDataInputParams params;

ret = UserProfile::Interface::getNpId(&npid, false);
if(ret < 0) {
    // Error handling
}

params.npid = *npid.get();
params.slotId = 1;
params.isVirtualUser = false;

ret = TUS::Interface::getData(&data, params, false);
if (ret < 0) {
    // Error handling
}
```

If `getData()` is called asynchronously, a `tusDataReceived` event will be sent to the NP Toolkit callback on successful completion.

Note: To perform this operation on a virtual user, assign true to `params.isVirtualUser` and add the virtual user's handle to `params.npid.handle`.

List of Functions

Table 26 Title User Storage Related Functions

Function	Description
`getData()`	Gets a specified user's TUS binary data.
`getVariables()`	Gets a specified user's TUS variables.
`setData()`	Sets a specified user's TUS binary data.
`setVariables()`	Sets a specified user's TUS variables.

17 Title Small Storage (TSS)

The title small storage (TSS) service supports the distribution of arbitrary data, game balance adjustments, and special events which are supplied after the title's release (this may be in the form of an announcement). The distributed data is held on a TSS server. When server maintenance is taken into account (server monitoring, domain name registration, and SSL certificate updates), using the TSS server is much easier than creating a server for one particular title. The following limitations exist for the data that can be distributed via the TSS service.

- One file on slot 0 of maximum size of 64KB.
- Fifteen files on slot 1 to 15 of up to 4MiB per slot.
- The file holding the data must be non-executable.
- The data in the file needs to have been deemed by SCE not to require QA.

The TSS service is available for the development environment, the QA environment, and the production environment. The only method available for transferring files to any of the TSS servers is scp. scp accounts and the required authentication files can be created with the PlayStation™Network Server Management Tools (SMT).

Getting TSS Data

Call `getData()` to obtain the data from the TSS server. Specify a Future object of the `TssData` type for the *`data`* parameter.

The following code example shows the procedure for obtaining TSS data for a title:

```
int ret;
Future<sce::Toolkit::NP::TssData> data;

ret = TSS::Interface::getData(&data);
if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

The above process is performed asynchronously. To obtain the results of the process, an application can either wait for the result or error set in the `Future` object, or it can wait for a `tssGotData`, `tssNoData` or `tssError` event to be returned to the NP Toolkit callback.

Note: This function only receives data from slot 0 on the TSS server, which has a maximum size of 64KB. To retrieve data from any slot, please use `getDataFromSlot()`.

Getting TSS Data Status

Call `getDataStatus()` to obtain the status of data from a specified slot on the TSS server. Specify a Future object of `SceNpTssDataStatus` type for the *`status`* parameter. With this you can query the size of the data on a specified slot and when it was last modified.

The following code example shows the procedure for obtaining TSS data status:

```
uint32_t slotId = 1;
Future<SceNpTssDataStatus> status;

int ret = TSS::Interface::getDataStatus(&status, slotId);
if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}
```

Getting TSS Data From a Slot

Call `getDataFromSlot()` to obtain the data from a specified slot on the TSS server. Specify a Future object of the `TssData` type for the *`data`* parameter. It is recommended that you call `getDataStatus()` first to get the size of the data on the specified slot. Use this size to create a buffer big enough to receive the data. You can also use the check the last modified data to determine if you even need to get the data again. The buffer that you allocate for `TssInputParams` must remain valid until after `getDataFromSlot()` has finished processing.

The following code example shows the procedure for obtaining TSS data for a title:

```
int ret;
uint32_t slotId = 0; // Can be between 0-15
Future<SceNpTssDataStatus> status;
Future<sce::Toolkit::NP::TssData> data;

ret = TSS::Interface::getDataStatus(&status, slotId, false);
if(ret != SCE_TOOLKIT_NP_SUCCESS)
{
    // Error handling
}

if (status.hasResult()) {

    uint8_t * buffer = new uint8_t[status.get()->contentLength];
    memset(buffer, 0x0, status.get()->contentLength);

    TssInputParams params;
    params.buffer = buffer;
    params.size = status.get()->contentLength;
    params.slotId = slotId;

    ret = TSS::Interface::getDataFromSlot(&data, params, false);
    if (ret < 0) {
        // Error handling
    } else if(g_tssData.hasResult()) {
        printf("Got %i bytes of data from TSS\n", g_tssData.get()->size);
        }

    // Cleanup when done
    delete [] params.buffer;
}
```

To obtain the results of the process, an application can either wait for the result or error set in the `Future` object, or it can wait for a `tssGotData`, `tssNoData` or `tssError` event to be returned to the NP Toolkit callback if called asynchronously.

List of Functions

Table 27 Title Small Storage Related Functions

Function	Description
`getData()`	Retrieves TSS data.
`getDataFromSlot()`	Retrieves TSS data from specified slot.
`getDataStatus()`	Retrieve status of data on specified slot.

18 Sessions

The NP Toolkit sessions interface provides an interface to create a session on the Session/Invitation server using the native interface and without using an NP Matching server. This is allows for sessions to be used with custom matchmaking services. For more information about the Session/Invitation server, please refer to the *Session/Invitation Overview* document.

Session Functionality

The main purpose of the interface is to provide an easier way of creating and managing session information for the application.

To join a session, you must either create a session yourself or join a session that was created by another user. The information about a session is contained within the `NpSessionInformation` structure. For more information about the parameters of the structure, please refer to the *NP Toolkit Library Reference*.

Creating a Session

Because the session interface omits the NP Matching server, the creation of a session using it differs slightly from the creation of a session using the matching interface. Specifically, `CreateNpSessionRequest` is used instead of `CreateSessionRequest` as an input parameter. The main difference here is that an amount of binary data of up to 1MiB in size can be sent as `sessionData` rather than NP Matching 2 specific parameters such as world ID or server ID.

It is also possible to add localized names for up to 10 additional localizations. If additional localized strings are not used, the default `sessionName` will be used. The system will display the string for the language that the user has specified in the system preferences.

In the NP Toolkit Sample, localizations can easily be added to the request by using the `addLocalizesdSessionName()` function which takes two arguments:

- A language code (up to 5 characters in length). For example, “en-GB” is for UK English.
- A localized session name for the language code (up to 64 characters in length).

The NP Toolkit Sample contains sample localizations.

To create a session, call `create()` and pass in these three parameters:

- `CreateNpSessionRequest *sessionRequest`
- `Future<NpSessionInformation> *sessionInformation`
- `bool async`

The `CreateNpSessionRequest` structure specifies the session parameters of the room.

Table 28 CreateNpSessionRequest Members

Structure Member	Description	Specification	Change After Creation?
sessionName	The session name.	Required	Possible
sessionStatus	The status string which will be registered with the session server.	Required	Possible
sessionImgPath	The path of the image to be uploaded to the session server.	Required	Possible
sessionData	The binary data which can be of a size of up to 1MiB.	Required	Possible
sessionDataSize	The size of the binary data	Required	Possible
sessionTypeFlag	A flag that specifies whether the session is private or public.	Required	Not possible

Structure Member	Description	Specification	Change After Creation?
sessionCreateFlag	A flag that specifies whether the session is "owner-bind" or "owner-migration"	Required	Not possible
maxSlots	The total number of slots (up to a maximum of 64).	Required	Not possible
localizedSessionName	Localized session names (up to a maximum of 10).	Not Required	Possible
sessionChangeableData	Changeable data attachment for the session.	Not required	Possible

Once the arguments have been set up, the application can call `Sessions::Interface::create()`.

If the request was kicked off successfully, the function will return `SCE_TOOLKIT_NP_SUCCESS`. The application will be notified by an `npSessionCreateResult` callback event when the create process has been completed. To check whether the session was successfully created or if there was an error, the application must check the `hasResult()` or `hasError()` of the `NpSessionInformation` Future object passed in. If a session is created successfully, the application can get the information about the session from the Future object `getData()` function.

The following code shows the procedure for creating a session:

```
sce::Toolkit::NP::CreateNpSessionRequest createSessionRequest;
static Future< sce::Toolkit::NP::NpSessionInformation> createFuture;

memset(&createSessionRequest, 0 ,
sizeof(sce::Toolkit::NP::CreateSessionRequest));

createSessionRequest.sessionTypeFlag =
sessionType;

ret = Sessions::Interface::create(&createSessionRequest,&createFuture);

if (ret < 0 ) {
   printf("Unable to create a request for session creation 0x%x\n",ret);
}
```

Once the session is created, the library will internally register a session on the session server, and the user that requested the session will automatically be a member of the created session. The *npSessionId* member of the `NpSessionInformation` Future object will contain the information about the session. The *errorCode* member can contain an error code if the session if the session failed to be created on the session server.

Joining a Session

A specific session can be joined by calling `Sessions::Interface::join()`. A `JoinNpSessionRequest` object needs to be prepared as an argument to pass to the function. Each `JoinNpSessionRequest` member can be considered to be a request parameter. These are shown below:

Table 29 Join Session Request Parameters

Request Parameter	Structure Member	Specification
Session ID	*npSessionId*	Required
Invitation Param	*invitationParam*	Optional
Invite Flag	*invite*	Optional

Before a session can be joined, the required *npSessionId* parameter must contain the ID of the session to be joined. In the NP Toolkit sample, a Friend's sessions can be searched in order to obtain the ID of a session to join. The ID is copied from the search results into the `JoinNpSessionRequest` object before `Sessions::Interface::join()` is called:

```
memcpy(&sessionReq.npSessionId,&searchListFuture.get()->front().npSessionId,
sizeof(sessionReq.npSessionId));
    int ret =
    sce::Toolkit::NP::Sessions::Interface::join
    (&sessionReq,&MenuApp::m_joinSession);
    if( ret< 0 ) {
        TTY::onScreenPrintf(MENU_TTY_TEXT_COLOUR_ERROR,"Failed to join NP
        Session 0x%x\n", ret);
        }
```

An `NpSessionInformation` Future object also needs to be passed as an argument. This receives the session information when a session has been successfully joined.

Searching for a Session

In the NP Toolkit Sample, before a session can be joined, you need to call `Sessions::Interface::search()`. Specify a `SearchNpSessionRequest` argument for this API function call. This structure specifies the following request parameters:

Table 30 Search Session Request Parameters

Request Parameter	Structure Member	Specification
onlineId	*SceNpOnlineId*	Required

Populate the `SearchNpSessionRequest` structure with the online ID of the user whose session is to be searched. In the following example, the online ID of the user to be searched is “LloydStemple”.

```
searchRequest.userInfo.userId = userId;
strncpy(searchRequest.onlineId.data,"LloydStemple",strlen("LloydStemple"));
```

Once the arguments have been set up, the application can call `Sessions::Interface::search()`. If the search request was kicked off successfully, the function will return `SCE_TOOLKIT_NP_SUCCESS`. The application is notified by an `npSessionSearchResult` callback event when the search has completed. The `NpSessionsList` Future object that was passed in is populated with the results of the search. `NpSessionsList` is a typedef, which represents a vector of `NpSessionInformation` objects.

Leaving a Session

A currently joined/created session can be left by calling `Sessions::Interface::leave()`. Specify a `LeaveNpSessionRequest` object to pass to this API function call. The `LeaveNpSessionRequest` structure contains the `SceNpSessionId` member shown below:

Table 31 Leave Session Request Parameters

Request Parameter	Structure Member	Specification
npSessionId	*SceNpSessionId*	Required

Populate the `LeaveNpSessionRequest` structure with the session ID of the session that is to be exited by the user. Once the arguments have been set up, the application can call `Sessions::Interface::leave()`. If the leave request was kicked off successfully, the function will return `SCE_TOOLKIT_NP_SUCCESS`. The application is notified by an `npSessionLeaveResult` callback event when the request to leave has been completed. The `int` Future object that was passed in is populated with a return value indicating the result of the leave request.

Table 32 Return Values in Leave Session Future int

Return Value	Description
`SCE_TOOLKIT_NP_SUCCESS`	The application has successfully left the session.
`SCE_TOOLKIT_NP_NOT_INITIALISED`	The operation failed because NP Toolkit library was not initialized.

Return Value	Description
SCE_TOOLKIT_NP_MATCHING_SERVICE_BUSY	The operation failed because the matching service is busy processing a previous request.
SCE_TOOLKIT_NP_MATCHING_SESSION_DOES_NOT_EXIST	The operation failed because the session the user which the user is trying to leave does not exist.

Inviting Users to a Session

A session invite can be sent to up to 16 users.

The example code in the NP Toolkit sample allows a friend to be invited to the session that the user is currently a member of.

This can be done by calling `Sessions::Interface::invite()`. Specify an `InviteNpSessionRequest` object and an `InviteMessage` object to pass to this API function call. Each of the following `InviteMessage` members can be considered to be a request parameter. These are shown below:

Table 33 Invite to Session Request Parameters

Structure Member	Description	Specification	Change After Creation?
iconPath	The path to the icon which needs to be displayed in the message.	Required	Not possible
body	The body text of the message.	Required	Not possible
expireMinutes	The amount of time until the expiration of the message in minutes from now.	Required	Not possible
padding	Padding	Required	Not possible

Once the arguments have been set up, the application can call `Sessions::Interface::invite()` to send the invite. If successful, the function will return `SCE_TOOLKIT_NP_SUCCESS`. If not successful, the function will return `SCE_TOOLKIT_NP_MATCHING_SESSION_DOES_NOT_EXIST`. This means that the operation failed because the invite that was sent was invalid.

Sending an Invitation with a Data Attachment

Arbitrary binary data can be attached to an invitation. This can be used to describe the params needed to join the session. This can be used to describe data required to join the user sending the invite.

```
sce::Toolkit::NP::PostInvitationDataRequest request;
char payload[10] = {'A','B','C','D','E','F','G','H','I','J'};

request.userInfo.userId = userId;
SceNpOnlineId onlindeId;
memset(&onlindeId,0,sizeof(onlindeId));
strncpy(onlindeId.data,"still_bird",strlen("still_bird"));
request.numOnlineIds = 1;
strncpy(request.npTitleId.id,NP_TITLE_ID,strlen(NP_TITLE_ID));
strncpy(request.message,"test",strlen("test"));
request.data = payload;
request.dataSize = 10;
request.onlineIds = &onlindeId;
memcpy(request.npSessionId.data,m_currentSession[0].npSessionId.data,sizeof(
m_currentSession[0].npSessionId.data));

ret = sce::Toolkit::NP::Sessions::Interface::postInvitationData(
&request,&result future);
if( ret < 0 ) {
    // handle error.
} else {
    // successfully kicked off.
}
```

When the request has been completed then an `npSessionInvitePostInvitationResult` event is raised on the call back and the provided Future will contain the error code if the request failed.

Getting Session and Invitation Data Attachments

Arbitrary binary data can be attached to both session and invitations. This can be used to describe params needed to join the session or actions that need to be taken when the user joins the session. To get session data attachments the following code is used:

```
sce::Toolkit::NP::GetInfoNpSessionRequest req;
memcpy(&req.npSessionId, targetSession, sizeof(req.npSessionId));
req.userInfo.userId = userId;
sce::Toolkit::NP::Utilities::Future<sce::Toolkit::NP::MessageAttachment>
sessionData;
int ret = sce::Toolkit::NP::Sessions::Interface::getSessionData(
   &req, &sessionData, false);
if(ret < 0) {
    // error handling
} else {
    // successfully completed request.
}
```

Once the request has been completed an `npSessionGetSessionDataResult` event is triggered on the main event callback thread and the `MessageAttachment` Future object will contain the error or the result of the call.

The following is an example of getting the data attachment of an invitation:

```
SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&userId);
if( ret < 0 ) {
    // error handling
    return;
}
sce::Toolkit::NP::InvitationDataRequest req;
memcpy(&req.invitationId, &invitations.get()->at(0).invitationId,
sizeof(req.invitationId));
req.userInfo.userId = userId;
sce::Toolkit::NP::Utilities::Future<sce::Toolkit::NP::MessageAttachment>
invitationData;
ret =  sce::Toolkit::NP::Sessions::Interface::getInvitationData(
    &req, &invitationData);
if (ret < 0) {
    // error handling
}
```

Once the request has been completed an `npSessionGetSessionDataResult` event is triggered on the main event callback thread and the `MessageAttachment` Future object will contain the error or the result of the call.

Set Invitation Data Used Flag

Each invitation has a data used flag, which allows an application to mark an invitation as used. Although this is not necessary as the system will maintain validity, when a invitation is marked as used, data cannot be retrieved from the invitation again. The following is an example of a call to set the data used flag:

```
sce::Toolkit::NP::InvitationDataRequest useFlagRequest;
```

```
sce::Toolkit::NP::Utilities::Future<int> useFlagResult;
useFlagRequest.userInfo.userId = userId;
memcpy(&useFlagRequest.invitationId, &invitations.get()->at(0).invitationId,
sizeOf(useFlagRequest.invitationId));
int ret = sce::Toolkit::NP::Sessions::Interface::setInvitationDataUsedFlag(
    &useFlagRequest, &useFlagResult);
if(ret < 0) {
    // error handling
} else {
    // completed request.
}
```

Once the call is completed a `npSessionInviteSetDataUsedResult` event will be raised on the main event callback thread.

Updating a Session

If a calling user is the session owner (the default in the NP Toolkit Sample) or the *`sessionCreateFlag`* is set to `SCE_TOOLKIT_NP_CREATE_MIGRATION_SESSION` to specify owner migration, then the session information can be updated.

The session information can be updated by calling `Sessions::Interface::update()`. This function requires three parameters:

- `UpdateNpSessionRequest` *`*updateSessionRequest`*
- `Future<int>` *`*processResult`*
- `bool` *`async`*

Calling this function will update the values contained within the *`updateSessionRequest`* argument.

Table 34 Update Session Request Parameters

Structure Member	Description	Specification	Change After Creation?
sessionName	The session name.	Required	Possible
sessionStatus	The status string which will be registered with the session server.	Required	Possible
sessionData	Specify binary data up to 1MiB.	N/A	Not Possible
sessoinDataSize	The size of the data.	N/A	Not Possible
npSessionId	The session ID related to the session server.	Required	Possible
sessionChangeableData	Changeable session data attachment.		Possible

Once the arguments have been set up, the application can call `Sessions::Interface::update()`. If the search request was kicked off successfully, the function will return `SCE_TOOLKIT_NP_SUCCESS`. The application is notified by an `npSessionUpdateResult` callback event when the update process has completed. The `int` Future object that was passed in is populated with a return value indicating the result of the update.

Note: If localizations were previously specified but not updated (e.g. only *`sessionName`* was updated) then they will be lost.

Table 35 Return Values in Update Session Future Int

Return Value	Description
`SCE_TOOLKIT_NP_SUCCESS`	The session information was updated successfully.
`SCE_TOOLKIT_NP_MATCHING_SESSION_KICKEDOUT`	The operation failed because the user has been kicked out of the matching session.

Return Value	Description
`SCE_TOOLKIT_NP_MATCHING_SERVICE_BUSY`	The operation failed because the matching service is busy processing a previous request.
`SCE_TOOLKIT_NP_MATCHING_SESSION_DOES_NOT_EXIST`	The operation failed because the session the user was trying to update does not exist.
`SCE_TOOLKIT_NP_MATCHING_SESSION_ROOM_DESTROYED`	The operation failed because the session the user was in has been destroyed.

Note: On receiving an error, the application should clear the current session.

Getting More Information About a Session

The application can retrieve detailed information about the session by calling `Sessions::Interface::getInfo()`. This function requires three parameters:

- `GetInfoNpSessionRequest` *`*sessionInfoRequest`*
- `Future<NpSessionDetailedInformation>` *`*sessionInformation`*
- `bool` *`async`*

Specify a `GetInfoNpSessionRequest` structure to pass as an argument to this API function call. The `GetInfoNpSessionRequest` structure contains the session ID of the target session from where the details should be obtained from:

Table 36 Get Info Np Session Request Parameters

Request Parameter	Structure Member	Specification
`npSessionId`	*`SceNpSessionId`*	Required

Once the arguments have been set up, the application can call `Sessions::Interface::getInfo()`. If the search request was kicked off successfully, the function will return `SCE_TOOLKIT_NP_SUCCESS`. The application is notified by an `npSessionGetInfoResult` callback event when the request for information has completed. The `NpSessionDetailedInformation` Future object that was passed in is populated with the detailed information about the session.

List of Functions

Table 37 Sessions Related Functions

Function	Description
`create()`	Creates a session on the NP Session server.
`getInfo()`	Get specific session information.
`invite()`	Sends a session invite to a friend of the user.
`join()`	Joins a specific session.
`leave()`	Leaves a currently joined/created session.
`search()`	Searches for a session.
`update()`	Updates the current session information.
`getInvitationInfo()`	Provided information about an invitation received by a user.
`getInvitationData()`	Retrieves data attached to an invitation.
`setInvitationDataUsedFlag()`	Sets the data used flag on invitation data.
`getInvitationList()`	Retrieves a list of invitations received by a user.
`getSessionData()`	Retrieves a session binary data attachment.
`getChangeableSessionData()`	Retrieves a session changeable binary data attachment.

19 Game Custom Data

The NP Toolkit game custom data interface provides a method to retrieve game custom data using a native interface from the PlayStation™Network server. For more information on game custom data, please refer to the *GameCustomDataDialog Library Overview* documentation.

Sending a Game Custom Data Message

In order to send a game custom data message, the application can make use of the NP Toolkit messaging interface and call `sendMessage()`. When calling `sendMessage()`, `SCE_TOOLKIT_NP_MESSAGE_TYPE_CUSTOM_DATA` must be passed to the *`messageType`* parameter to specify that a game custom data message is being sent.

As required, the `GameCustomDataDialog` can also be shown to allow the user to input the message. This is achieved by specifying `SCE_TOOLKIT_NP_DIALOG_TYPE_USER_EDITABLE` for the *`dialogFlag`* member of the `MessageData` object passed to `sendMessage()`.

Up to 1MiB of custom data to another user using `sendMessage()`. The following code sample shows this:

```
sce::Toolkit::NP::MessageData messData;
#define MESSAGE_ATTACHMENT “Binary data attachment”
SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&userId);

if( ret < 0 ) {
    TTY::onScreenPrintf(MENU_TTY_TEXT_COLOUR_ERROR,
    "Couldn't retrieve user ID 0x%x ...\n",ret);
        }
char buffer[100];
memset(buffer,0,sizeof(buffer));
strncpy(buffer,MESSAGE_ATTACHMENT ,strlen(MESSAGE_ATTACHMENT ));

messData.attachment = buffer;
messData.attachmentSize = sizeof(buffer);
messData.body.assign("Test Custom Data ");
messData.dialogFlag = SCE_TOOLKIT_NP_DIALOG_TYPE_USER_EDITABLE;
messData.npIdsCount = 2;
messData.npIds = NULL;
messData.userInfo.userId = userId;
messData.attachment = buffer;
messData.attachmentSize = strlen(buffer);
messData.dataDescription.assign("Toolkit Test Custom Data");
messData.dataName.assign("Custom data");
messData.iconPath.assign(SESSION_IMAGE_PATH);

ret = sce::Toolkit::NP::Messaging::Interface::sendMessage
(&messData,SCE_TOOLKIT_NP_MESSAGE_TYPE_CUSTOM_DATA);

if( ret < SCE_TOOLKIT_NP_SUCCESS ) {
              return ;
        }
```

Retrieving Game Custom Data Items

The NP Toolkit custom game data interface provides the `getItemList()` method to retrieve a list of received game custom data items from the PlayStation™Network server. The following code sample shows this:

```
sce::Toolkit::NP::Utilities::Future<sce::Toolkit::NP::GameCustomDataItemList
> msgAttach;
sce::Toolkit::NP::GameCustomDataItemsRequest requestInfo;
SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;

int ret = sceUserServiceGetInitialUser(&userId);
if( ret < 0 ) {
    //Error handling
}

requestInfo.userInfo.userId = userId;

ret =
sce::Toolkit::NP::GameCustomData::Interface::getItemList(&requestInfo,&msgAt
tach,false);

if( ret < 0 ) {
    //Error handling
} else {
    TTY::onScreenPrintf(MENU_TTY_TEXT_COLOUR, "Custom Data Retrieved\n",ret);
}
```

Retrieving Game Custom Data

The NP Toolkit custom game data interface provides the `getGameData()` method to retrieve game data that is attached to a custom game data message. The following code sample shows this:

```
sce::Toolkit::NP::GameDataRequest req;
SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&userId);
if( ret < 0 ) {
    //Error handling
}

    req.itemId = ItemID ;
    req.userInfo.userId = userId;
    sce::Toolkit::NP::Utilities::Future<sce::Toolkit::NP::MessageAttachment>
    actualMessage;
    ret = sce::Toolkit::NP::GameCustomData::Interface::getGameData
    (&req,&actualMessage,false);
if( ret < 0 ) {
    //Error handling
} else {
    //Error handling
}
```

Retrieving a Game Custom Data's Thumbnail

Each game custom data message is able to have an attached JPEG thumbnail, which is used in system software and optionally can be used within an application. The following shows an example of how to obtain this:

```
sce::Toolkit::NP::Utilities::Future<sce::Toolkit::NP::MessageAttachment>
gcdThumb
sce::Toolkit::NP::GameCustomDataThumbnailRequest req;

req.userInfo.userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&req.userInfo.userId); // get the userID
if( ret < SCE_OK ) {
    // error handling
    return;
```

```
        }

        req.itemId = SELECTED_ITEM_ID;

        ret = sce::Toolkit::NP::GameCustomData::Interface::getThumbnail(&req,
        &gcdThumb);
        if ( ret < SCE_TOOLKIT_NP_SUCCESS ) {
            // error handling
        }
```

When this request is completed a `gameCustomDataGameThumbnailResult` event is raised on the callback. The `MessageAttachment` Future object will then contain either the error code or the result of the request.

List of Functions

Table 38 Game Custom Data Related Functions

Function	Description
`getItemList()`	Gets a list of game custom data items.
`getGameData()`	Gets the game data attached to a received custom game data message.
`getMessage()`	Gets a game custom data message.
`setMessageUseFlag()`	Sets the use flag for a game custom data message.
`getThumbnail()`	Gets the thumbnail image attached to a received game custom data message.

20 Challenges

The NP Toolkit Challenges interface provides a method to transfer messages with binary attachments between users. The intended use of these messages is to send and receive gameplay related binary data in the form of challenges. Challenges should be considered as a list of gameplay related conditions that the recipient must achieve in order to succeed.

The interface has the ability to send, receive and reply to challenges with a number of options.

The Challenges interface is a convenience API that has been built using the Game Custom Data PlayStation™Network service. Registering for this service is required before use of the Challenges interface.

Use of this API is exemplified in the NP Toolkit Sample. The Challenges interface does not use Common Dialogs and therefore all interaction is achieved programmatically without a GUI.

Sending a Challenge

In order to send a challenge, the application must prepare several pieces of metadata. These consist of text that is displayed to the user. The text can be seen in the system GUI within the Game Alerts section of the Notifications area. The text should contain a name and a description that describes the pass condition of the challenge. A user message can be optionally added. Up to `SCE_TOOLKIT_NP_CHALLENGES_SEND_ATTACHMENT_MAX_SIZE` of extra binary data can also be sent using `sendChallenge()`.

The following code sample shows `sendChallenge()` in use:

```
SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&userId);
if( ret < 0 ) {
    /* Error Handling */
}
SceNpId npId;
ret = sceNpGetNpId(userId, &npId);
if(ret < 0){
    /* Error Handling */
}
ChallengeSendRequest request;
request.userInfo.userId = userId;
request.recipients = &npId.handle; // Sending it to ourselves
request.recipientsNum = 1;
request.imagePath = SESSION_IMAGE_PATH;
request.dataSize = sizeof(CUSTOM_DATA_ATTACHMENT);
request.expiresMins = 20;
request.data = const_cast<SceChar8*>(CUSTOM_DATA_ATTACHMENT);
strncpy(request.userMessage, "Just had a wicked lap on track 4! Just try and beat
it!", sizeof(request.userMessage) -1);
strncpy(request.description, "Complete Track 4 under the following
conditions:\\n - Finish 1st\\n - Finish without boosting\\n - Power slide for
400m total\\n\\nReward:\\n - 800 credits", sizeof(request.description)-1);
strncpy(request.name, "Challenge Name", sizeof(request.name) -1);
//Optional supported languages (empty)

request.localizedMetadata = NULL;
request.localizedMetadataNum = 0;
Utilities::Future<SendChallengeResult> output;
ret = Challenges::Interface::sendChallenge(request, &output, false);
if( ret < SCE_TOOLKIT_NP_SUCCESS ) {
    /* Error Handling */
}
```

Retrieving Inbox Items

The NP Toolkit Challenges interface provides a `getItemList()` method to retrieve a list of received challenge related items from the PlayStation™Network server. This function handles both the retrieval of new challenges as well as challenge responses. The option is available to retrieve challenges, responses to challenges or both. This function does not retrieve a challenge's binary data attachment. Please use `getChallengeData()` to obtain the binary data. In addition to retrieving a list, it is also possible to retrieve a specific challenge item using `getItem()`.

The following code sample shows `getItemList()` in use:

```
SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&userId);
if( ret < 0 ) {
    /* Error Handling */
}

ChallengeGetItemListRequest request;
request.userInfo.userId = userId;
request.numChallengesToGet = 5;
request.typeToGet = SCE_TOOLKIT_NP_CHALLENGES_RETRIEVE_TYPE_CHALLENGE;
request.filterUnusable = false;

Utilities::Future<ReceivedChallengeList> inboxList;
ret = Challenges::Interface::getItemList(request, inboxList, false);
if( ret < SCE_TOOLKIT_NP_SUCCESS ) {
    /* Error Handling */
}
```

Retrieving Binary Data Attachments

The NP Toolkit Challenges interface provides the `getChallengeData()` method to retrieve challenge binary data from a challenge message. After retrieving a challenge using `getItemList()`, assign the `ChallengeRecvDetails::inboxId` to `ChallengeGetDataRequest::inboxId` before making the call.

The following code sample shows `getChallengeData()` in use:

```
SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&userId);
if( ret < 0 ) {
    /* Error Handling */
}

//Previously obtained Challenge list from getItemList()
ChallengeRecvDetails challenge = (*g_receivedChallengeList.get())[0];
ChallengeGetDataRequest dataRequest;
dataRequest.inboxId = challenge.inboxId;
dataRequest.userInfo.userId = userId;
Utilities::Future<ChallengeBinaryDataResult> result;
ret = Challenges::Interface::getChallengeData(dataRequest, &result, false);
if( ret < 0 ) {
    /* Error Handling */
}
```

Replying to a Received Challenge

Once the application has retrieved a challenge, the user can then attempt the challenge. The exact behaviour of this is application specific, and the Challenges interface does not define how this should be achieved. Once the application has determined the result of this attempt, a notification can be sent back to the original sender. In addition to this, a notification can also be sent at other times, such as when the challenge is initially accepted.

These notifications are not sent automatically and sending them is at the discretion of the application. They are displayed in the Game Alerts section of the Notifications area, and the user will receive an on-screen alert when they are received.

The following code sample shows `sendResponse()` in use:

```
SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&userId);
if( ret < 0 ) {
    /* Error Handling */
}

ChallengeReponseRequest request;
request.imagePath           = SESSION_IMAGE_PATH;
request.originalChallenge    = &(*g_receivedChallengeList.get())[0];
request.userMessage         = "I've accepted your challenge!";
request.status              = ChallengeStatus::ChallengeAccepted;
request.userInfo.userId      = userId;

Utilities::Future<NotifyChallengeResult> output;
ret = Challenges::Interface::sendResponse(request, &output, false);
if( ret < SCE_TOOLKIT_NP_SUCCESS ) {
    /* Error Handling */
}
```

Marking Challenge Binary Data as Used

Dependent on the style of challenge implementation in an application, it may be desirable to disallow further attempts of a challenge once it has been either passed or failed. In those circumstances, the application can use the `consumeItem()` method in order to notify the PlayStation™Network servers that the challenge is no longer available.

The following code sample shows `consumeItem()` in use:

```
SceUserServiceUserId userId = SCE_USER_SERVICE_USER_ID_INVALID;
int ret = sceUserServiceGetInitialUser(&userId);
if( ret < 0 ) {
    /* Error Handling */
}

// Previously obtained challenge list
ChallengeRecvDetails challenge = (*g_receivedChallengeList.get())[0];
ChallengeConsumeRequest consumeRequest;
consumeRequest.inboxId        = challenge.inboxId;
consumeRequest.userInfo.userId = userId;

Utilities::Future<ConsumeChallengeResult> result;
ret = Challenges::Interface::consumeItem(consumeRequest, &result, false);
if( ret < SCE_TOOLKIT_NP_SUCCESS ) {
    /* Error Handling */
}
```

List of Functions

Table 39 Challenges Related Functions

Function	Description
`sendChallenge()`	Sends a challenge to a user/users.
`getItemList()`	Gets a list of challenges the user has in their inbox.
`getChallengeData()`	Retrieves a challenge's data.
`sendResponse()`	Replies to a challenge.
`consumeItem()`	Consumes a challenge.
`getItem()`	Retrieves a single challenge related item.