TeamSpeak 3 Server
SDK Developer Manual

Revision 2012-07-19 12:50:53

Table of Contents

Copyright

TeamSpeak Systems GmbH
Soiernstrasse 1
82494 Krün
Germany

Visit TeamSpeak-Systems on the web at www.teamspeak.com [http://www.teamspeak.com]

License agreement

TeamSpeak 3

LICENSE AGREEMENT

October 25th, 2007

THIS IS A LEGAL AGREEMENT between "you," the company or end user of TeamSpeak 3 brand software, and TeamSpeak Systems GmbH, a Krün, Germany company hereafter referred to as "TeamSpeak Systems".

Use of the software you are about to install indicates your acceptance of these terms. You also agree to accept these terms by so indicating at the appropriate screen, prior to the download or installation process. As used in this Agreement, the capitalized term "Software" means the TeamSpeak 3 voice over IP (VoIP) communication software together with any and all enhancements, upgrades, and updates that may be provided to you in the future by TeamSpeak Systems. IF YOU DO NOT AGREE TO THESE TERMS AND CONDITIONS, YOU SHOULD SO INDICATE BY CONTACTING TEAMSPEAK SYSTEMS AND PROMPTLY DISCONTINUE THE INSTALLATION PROCESS AND USE OF THIS SOFTWARE.

Ownership

The Software and any accompanying documentation are owned by TeamSpeak Systems and ownership of the Software shall at all times remain with TeamSpeak Systems. Copies are provided to you only to allow you to exercise your rights under this Agreement. This Agreement does not constitute a sale of the Software or any accompanying documentation, or any portion thereof. Without limiting the generality of the foregoing, you do not receive any rights to any patents, copyrights, trade secrets, trademarks or other intellectual property rights relating to or in the Software or any accompanying documentation. All rights not expressly granted to you under this Agreement are reserved by TeamSpeak Systems.

Grant of License Applicable To TeamSpeak 3

Subject to the terms and conditions set out in this Agreement, TeamSpeak Systems grants you a limited, nonexclusive, nontransferable and nonsublicensable right to use the Software called "TeamSpeak 3" solely in accordance with the following terms and conditions:

1. Use of TeamSpeak 3. You may use TeamSpeak 3 on multiple computers owned, leased or rented by you, your company, or business entity; however, you are the only individual, company, or business entity with the right to use your licensed copy(ies) of TeamSpeak 3. All copies of TeamSpeak 3 must include TeamSpeak Systems' copyright notice.

2. Distribution Prohibited. You may not distribute copies of TeamSpeak 3 for use by anyone other than you, your company, or business entity. Distribution of TeamSpeak 3 by you to third parties is hereby expressly prohibited.

3. Fees. As of the date listed above for this License Agreement, TeamSpeak 3 is in a "pre-release" stage. Fees and licensing costs will be determined when the final version of the product is released or an agreed upon commencement date for commercial use of the Software is initiated.

4. Termination. TeamSpeak Systems may terminate your TeamSpeak 3 license at any time, for any reason or no reason. TeamSpeak Systems may also terminate your TeamSpeak 3 license if you breach any of the terms and conditions set forth in this Agreement. Upon termination, you shall immediately destroy all copies of TeamSpeak 3 and any accompanying files or documentation in your possession, custody or control.

5. Support. TeamSpeak Systems will provide you with support services related to TeamSpeak 3 for a period that begins on the date TeamSpeak 3 is delivered to you, and ends upon the termination of this Agreement.

6. Upgrades. TeamSpeak Systems will provide you with upgrades to TeamSpeak 3 for a period that begins on the date TeamSpeak 3 is delivered to you. Such upgrades will be released only by TeamSpeak Systems for the purpose of improving TeamSpeak 3 software. TeamSpeak Systems has no obligation to provide you with any upgrades that are not released for general distribution to TeamSpeak Systems' other licensees. Nothing in this Agreement shall be construed to obligate TeamSpeak Systems to provide upgrades to you under any circumstances.

Prohibited Conduct

You represent and warrant that you will not violate any of the terms and conditions set forth in this Agreement and that:

You will not, and will not permit others to: (i) reverse engineer, decompile, disassemble, derive the source code of, modify, or create derivative works from the Software; or (ii) use, copy, modify, alter, or transfer, electronically or otherwise, the Software or any of the accompanying documentation except as expressly permitted in this Agreement; or (iii) redistribute, sell, rent, lease, sublicense, or otherwise transfer rights to the Software whether in a stand-alone configuration or as incorporated with other software code written by any party except as expressly permitted in this Agreement.

You will not use the Software to engage in or allow others to engage in any illegal activity.

You will not engage in use of the Software that will interfere with or damage the operation of the services of third parties by overburdening/disabling network resources through automated queries, excessive usage or similar conduct.

You will not use the Software to engage in any activity that will violate the rights of third parties, including, without limitation, through the use, public display, public performance, reproduction, distribution, or modification of communications or materials that infringe copyrights, trademarks, publicity rights, privacy rights, other proprietary rights, or rights against defamation of third parties.

You will not transfer the Software or utilize the Software in combination with third party software authored by you or others to create an integrated software program which you transfer to unrelated third parties.

Upgrades, Updates And Enhancements

All upgrades, updates or enhancements of the Software shall be deemed to be part of the Software and will be subject to this Agreement.

Disclaimer of Warranty

THE SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE WARRANTIES THAT IT IS FREE OF DEFECTS, VIRUS FREE, ABLE TO OPERATE ON AN UNINTERRUPTED BASIS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE AND AGREEMENT. NO USE OF THE SOFTWARE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

Limitation of Liability

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL TEAMSPEAK SYSTEMS BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF

THE USE OF OR INABILITY TO USE THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOST PROFITS, LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF, AND REGARDLESS OF THE LEGAL OR EQUITABLE THEORY (CONTRACT, TORT OR OTHERWISE) UPON WHICH THE CLAIM IS BASED. IN ANY CASE, TEAMSPEAK SYSTEMS' COLLECTIVE LIABILITY UNDER ANY PROVISION OF THIS LICENSE SHALL NOT EXCEED IN THE AGGREGATE THE SUM OF THE FEES (IF ANY) YOU PAID FOR THIS LICENSE.

Legends and Notices

You agree that you will not remove or alter any trademark, logo, copyright or other proprietary notices, legends, symbols or labels in the Software or any accompanying files or documentation.

Term and Termination

This Agreement is effective upon your acceptance as provided herein and payment of the applicable license fees (if any), and will remain in force until terminated. You may terminate the licenses granted in this Agreement at any time by contacting TeamSpeak Systems in writing, and destroying the Software and any accompanying files or documentation, together with any and all copies thereof. The licenses granted in this Agreement will terminate automatically if you breach any of its terms or conditions or any of the terms or conditions of any other agreement between you and TeamSpeak Systems. Upon termination, you shall immediately destroy the original and all copies of the Software and any accompanying documentation, or return them to TeamSpeak Systems.

Software Suggestions

TeamSpeak Systems welcomes suggestions for enhancing the Software and any accompanying documentation that may result in computer programs, reports, presentations, documents, ideas or inventions relating or useful to TeamSpeak Systems' business. You acknowledge that all title, ownership rights, and intellectual property rights concerning such suggestions shall become the exclusive property of TeamSpeak Systems and may be used for its business purposes in its sole discretion without any payment or accounting to you.

Miscellaneous

This Agreement constitutes the entire agreement between the parties concerning the Software, and may be amended only by a writing signed by both parties. This Agreement shall be governed by the laws of Krün, Germany, excluding its conflict of law provisions. All disputes relating to this Agreement are subject to the exclusive jurisdiction of the courts within Germany and you expressly consent to the exercise of personal jurisdiction in the courts of Germany in connection with any such dispute. This Agreement shall not be governed by the United Nations Convention on Contracts for the International Sale of Goods. If any provision in this Agreement should be held illegal or unenforceable by a court of competent jurisdiction, such provision shall be modified to the extent necessary to render it enforceable without losing its intent, or severed from this Agreement if no such modification is possible, and other provisions of this Agreement shall remain in full force and effect. A waiver by either party of any term or condition of this Agreement or any breach thereof, in any one instance, shall not waive such term or condition or any subsequent breach thereof.

Introduction

TeamSpeak 3 is a scalable Voice-Over-IP application consisting of client and server software. TeamSpeak is generally regarded as the leading VoIP system offering a superior voice quality, scalability and usability.

The cross-platform Software Development Kit allows the easy integration of the TeamSpeak client and server technology into own applications.

This document describes server-side programming with the TeamSpeak 3 SDK. The SDK user will be able to create a custom TeamSpeak 3 server binary using the provided server API and library.

System requirements

For developing third-party clients with the TeamSpeak 3 Server Lib the following system requirements apply:

- Windows

  Windows 2000, XP, Vista (32- and 64-bit)

- Mac OS X

  Mac OS X 10.4 and above on Intel and PowerPC

- Linux

  Any recent Linux distribution with libstdc++ 6. Both 32- and 64-bit are supported.

Important

The calling convention used in the functions exported by the shared TeamSpeak 3 SDK libaries is *cdecl*. You must not use another calling convention, like stdcall on Windows, when declaring function pointers to the TeamSpeak 3 SDK libraries. Otherwise stack corruption at runtime may occur.

Usage

All the required files are located in the `bin` directory of the TeamSpeak 3 SDK distribution.

Important

The license file `licensekey.dat` needs to be located in the same folder as your server executable.

If no license key is present, the server will run with the following limitations:

- Only one server process per machine
- Only one virtual server per process
- Only 32 slots

For more detailed information about licensing of TeamSpeak 3 servers or to obtain a license, please contact `<sales@tritoncia.com>`.

Calling Server lib functions

Server Lib functions follow a common pattern. They always return an error code or *ERROR_ok* on success. If there is a result variable, it is always the last variable in the functions parameters list.

```
ERROR ts3server_FUNCNAME(arg1, arg2, ..., &result);
```

Result variables should *only* be accessed if the function returned *ERROR_ok*. Otherwise the state of the result variable is undefined.

In those cases where the result variable is a basic type (int, float etc.), the memory for the result variable has to be declared by the caller. Simply pass the address of the variable to the Server Lib function.

```
int result;

if(ts3server_XXX(arg1, arg2, ..., &result) == ERROR_ok) {
    /* Use result variable */
} else {
    /* Handle error, result variable is undefined */
}
```

If the result variable is a pointer type (C strings, arrays etc.), the memory is allocated by the Server Lib function. In that case, the caller has to release the allocated memory later by using `ts3server_freeMemory`. It is important to *only* access and release the memory if the function returned *ERROR_ok*. Should the function return an error, the result variable is uninitialized, so freeing or accessing it could crash the application.

```
char* result;

if(ts3server_XXX(arg1, arg2, ..., &result) == ERROR_ok) {
    /* Use result variable */
    ts3server_freeMemory(result);  /* Release result variable */
} else {
    /* Handle error, result variable is undefined. Do not access or release it. */
}
```

Note

Server Lib functions are *thread-safe*. It is possible to access the Server Lib from several threads at the same time.

Initializing

When starting the server application, initialize the Server Lib with

```
unsigned int ts3server_initServerLib(functionPointers, usedLogTypes, logFileFolder);

const struct ServerLibFunctions* functionPointers;
int usedLogTypes;
const char* logFileFolder;
```

Note

This function must not be called more than once.

Parameters

- *functionPointers*

  Callback function pointers. See below.

- *usedLogTypes*

  Defines the log output types. The Server Lib can output log messages to a file (located in the `logs` directory relative to the server executable), to stdout or to user defined callbacks. If user callbacks are activated, the `onUserLoggingMessageEvent` event needs to be implemented.

  Available values are defined by the enum LogTypes (see `public_definitions.h`):

```
enum LogTypes {
    LogType_NONE          = 0x0000,
    LogType_FILE          = 0x0001,
    LogType_CONSOLE       = 0x0002,
    LogType_USERLOGGING   = 0x0004,
    LogType_NO_NETLOGGING = 0x0008,
    LogType_DATABASE      = 0x0010,
};
```

  Multiple log types can be combined with a binary OR. If only *LogType_NONE* is used, local logging is disabled.

Note

Logging to console can slow down the application on Windows. Hence we do not recommend to log to the console on Windows other than in debug builds.

Note

LogType_NO_NETLOGGING is no longer used. Previously this controlled if the Server Lib would send warning, error and critical log entries to a webserver for analysis. As netlogging does not occur anymore, this flag has no effect anymore.

LogType_DATABASE is unused in SDK builds.

- *logFileFolder*

  Location where the logfiles produced if file logging is enabled will be saved to. Pass NULL for the default behaviour, which is to use a folder called `logs` in the current working directory.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

The callback mechanism

The communication from the Server Lib to the server application takes place using callbacks. The server application has to define a series of function pointers using the struct ServerLibFunctions (see `serverlib.h`). These callbacks are used to let the server application hook into the library and receive notifaction on certain actions.

A callback example in C:

```
static void my_onClientConnected_callback(uint64 serverID, anyID clientID, uint64 channelID,
                                          unsigned int* removeClientError) {
    printf("Client %u connected on virtual server %u joining channel %u", clientID, serverID, channelID);
```

```
}
```

C++ developers can also use static member functions for the callbacks.

Before calling `ts3server_initServerLib`, create an instance of struct ServerLibFunctions, initialize all function pointers with NULL and point the structs function pointers to your implemented callback functions:

```
unsigned int error;

/* Create struct */
ServerLibFunctions slFuncs;

/* Initialize all function pointers with NULL */
memset(&slFuncs, 0, sizeof(struct ServerLibFunctions));

/* Assign those function pointers you implemented */
slFuncs.onVoiceDataEvent           = my_onVoiceDataEvent_callback;
slFuncs.onClientStartTalkingEvent  = my_onClientStartTalkingEvent_callback;
slFuncs.onClientStopTalkingEvent   = my_onClientStopTalkingEvent_callback;
slFuncs.onClientConnected          = my_onClientConnected_callback;
slFuncs.onClientDisconnected       = my_onClientDisconnected_callback;
slFuncs.onClientMoved              = my_onClientMoved_callback;
slFuncs.onChannelCreated           = my_onChannelCreated_callback;
slFuncs.onChannelEdited            = my_onChannelEdited_callback;
slFuncs.onChannelDeleted           = my_onChannelDeleted_callback;
slFuncs.onServerTextMessageEvent   = my_onServerTextMessageEvent_callback;
slFuncs.onChannelTextMessageEvent  = my_onChannelTextMessageEvent_callback;
slFuncs.onUserLoggingMessageEvent  = my_onUserLoggingMessageEvent_callback;
slFuncs.onAccountingErrorEvent     = my_onAccountingErrorEvent_callback;
slFuncs.onCustomPacketEncryptEvent = NULL;  // Not used by your application
slFuncs.onCustomPacketDecryptEvent = NULL;  // Not used by your application

/* Initialize library with callback function pointers */
error = ts3server_initServerLib(&slFuncs, LogType_FILE | LogType_CONSOLE);
if(error != ERROR_ok) {
    printf("Error initializing serverlib: %d\n", error);
    (...)
}
```

Important

As long as you initialize unimplemented callbacks with NULL, the Server Lib won't attempt to call those function pointers. However, if you leave unimplemented callbacks undefined, the Server Lib will crash when trying to call them.

The individual callbacks are described in the chapter Events.

Querying the library version

The Server Lib version can be queried with

```
unsigned int ts3server_getServerLibVersion(result);

char** result;
```

Parameters

- *result*

Address of a variable that receives the serverlib version string, encoded in UTF-8.

Caution

The result string must be released using `ts3server_freeMemory`. If an error has occured, the result string is uninitialized and must not be released.

To get only the version number, which is a part of the complete version string, as numeric value:

```
unsigned int ts3server_getServerLibVersionNumber(result);
```

```
uint64* result;
```

Parameters

- *result*

  Address of a variable that receives the numeric serverlib version.

Both functions return *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Example code to query the Server Lib version:

```
unsigned int error;
char* version;
error = ts3server_getServerLibVersion(&version);
if(error != ERROR_ok) {
    printf("Error querying serverlib version: %d\n", error);
    return;
}
printf("Server library version: %s\n", version);  /* Print version */
ts3server_freeMemory(version);  /* Release string */
```

Shutting down

Before exiting the application, the Server Lib should be shut down with

```
unsigned int ts3server_destroyServerLib();
```

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Any call to Server Lib functions after shutting down has undefined results.

Caution

Never destroy the Server Lib from within a callback function. This might result in a segmentation fault.

Error handling

Each Server Lib function returns either *ERROR_ok* on success or an error value as defined in `public_errors.h` if the function fails.

The returned error codes are organized in groups, where the first byte defines the error group and the second the count within the group: The naming convention is ERROR_<group>_<error>, for example *ERROR_client_invalid_id*.

Example:

```
unsigned int error;
char* welcomeMsg;

/* welcomeMsg memory is allocated if error is ERROR_ok */
error = ts3server_getVirtualServerVariableAsString(serverID, VIRTUALSERVER_WELCOMEMESSAGE, &welcomeMsg);
if(error != ERROR_ok) {
    /* Handle error */
    return;
}
/* Use welcomeMsg... */
ts3server_freeMemory(welcomeMsg);  /* Release memory *only* if function did not return an error */
```

Note

Result variables should *only* be accessed if the function returned *ERROR_ok*. Otherwise the state of the result variable is undefined.

Important

Some Server Lib functions dynamically allocate memory which has to be freed by the caller using `ts3server_freeMemory`. It is important to *only* access and release the memory if the function returned *ERROR_ok*. Should the function return an error, the result variable is uninitialized, so freeing or accessing it will likely result in a segmentation fault.

See the section Calling Server Lib functions for additional notes and examples.

A printable error string for a specific error code can be queried with

```
unsigned int ts3server_getGlobalErrorMessage(errorCode, error);

unsigned int errorCode;
char** error;
```

Parameters

- *errorCode*

  The error code returned from all Server Lib functions.

- *error*

Address of a variable that receives the error message string, encoded in UTF-8 format. Unless the return value of the function is not *ERROR_ok*, the string should be released with `ts3server_freeMemory`.

Example:

```
unsigned int error;
char* version;

error = ts3server_getServerLibVersion(&version);  /* Calling some Server Lib function */
if(error != ERROR_ok) {
    char* errorMsg;
    if(ts3server_getGlobalErrorMessage(error, &errorMsg) == ERROR_ok) {  /* Query printable error */
        printf("Error querying client ID: %s\n", errorMsg);
 ts3server_freeMemory(errorMsg);  /* Release memory only if function succeeded */
    }
}
```

Query virtual servers, clients and channels

A list of all virtual servers can be queried with:

```
unsigned int ts3server_getVirtualServerList(result);

uint64** result;
```

Parameters

- *result*

  Address of a variable which receives a NULL-terminated array of server IDs. Unless an error occured, the array should be released with `ts3server_freeMemory`.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. If an error has occured, the result array is uninitialized and must not be released.

Note

The default virtual server has an ID of 1.

A list of all clients currently online on the specified virtual server can be queried with:

```
unsigned int ts3server_getClientList(serverID, result);

uint64 serverID;
anyID** result;
```

Parameters

- *serverID*

ID of the virtual server on which the client list is requested.

- *result*

  Address of a variable which receives a NULL-terminated array of client IDs. Unless an error occured, the array should be released with ts3server_freeMemory.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If an error has occured, the result array is uninitialized and must not be released.

A list of all channels currently available on the specified virtual server can be queried with:

```
unsigned int ts3server_getChannelList(serverID, result);

uint64 serverID;
uint64** result;
```

Parameters

- *serverID*

  ID of the virtual server on which the channel list is requested.

- *result*

  Address of a variable which receives a NULL-terminated array of channel IDs. Unless an error occured, the array should be released with ts3server_freeMemory.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If an error has occured, the result array is uninitialized and must not be released.

To get a list of all clients currently member of the specified channel:

```
unsigned int ts3server_getChannelClientList(serverID, channelID, result);

uint64 serverID;
uint64 channelID;
anyID** result;
```

Parameters

- *serverID*

  ID of the virtual server on which the list of clients is requested.

- *channelID*

ID of the specified channel.

- *result*

  Address of a variable which receives a NULL-terminated array of client IDs. Unless an error occured, the array should be released with ts3server_freeMemory.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If an error has occured, the result array is uninitialized and must not be released.

Query the channel the specified client has currently joined:

```
unsigned int ts3server_getChannelOfClient(serverID, clientID, result);
```

```
uint64 serverID;
anyID clientID;
uint64* result;
```

Parameters

- *serverID*

  ID of the virtual server on which the channel is requested.

- *channelID*

  ID of the specified client.

- *result*

  Address of a variable which receives the ID of the channel the specified client has currently joined.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Get the parent channel of a given channel:

```
unsigned int ts3server_getParentChannelOfChannel(serverID, channelID, result);
```

```
uint64 serverID;
uint64 channelID;
uint64* result;
```

Parameters

- *serverID*

  ID of the virtual server on which the parent channel is requested.

- *channelID*

  ID of the channel whose parent channel is requested.

- *result*

  Address of a variable which receives the ID of the parent channel.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Example to print a list of all channels on a virtual server:

```
uint64* channels;

if(ts3server_getChannelList(serverID, &channels) == ERROR_ok) {
    for(int i=0; channels[i] != NULL; i++) {
        printf("Channel ID: %u\n", channels[i]);
    }
    ts3server_freeMemory(channels);
}
```

Example to print all clients who are member of channel with ID 123:

```
uint64 channelID = 123;  /* ID in our example */
anyID* clients;

if(ts3server_getChannelClientList(serverID, channelID, &clients) == ERROR_ok) {
    for(int i=0; clients[i] != NULL; i++) {
        printf("Client ID: %u\n", clients[i]);
    }
  ts3server_freeMemory(clients);
}
```

Create and stop virtual servers

A new virtual server can be created within the current server process by calling:

```
unsigned int ts3server_createVirtualServer(serverPort, serverIp, serverName, serverKeyPair, serverMaxClients, result);
```

```
unsigned int serverPort;
const char* serverIp;
const char* serverName;
const char* serverKeyPair;
unsigned int serverMaxClients;
uint64* result;
```

Parameters

- *serverPort*

  UDP port to be used for the new virtual server. The default TeamSpeak 3 port is UDP 9987.

- *serverIp*

IP to bind the virtual server to. Pass “0.0.0.0” to bind the virtual server to all IP addresses.

- *serverName*

  Name of the new virtual server. This can be later accessed through the virtual server property *VIRTUALSERVER_NAME*.

- *serverKeyPair*

  Unique keypair of this server. The first time you start this virtual server, pass an empty string, query the keypair with ts3server_getVirtualServerKeyPair, then save the keypair locally and pass it the next time as parameter to this function.

- *serverMaxClients*

  Maximum number of clients (“slots”) which can simultaneously be connected to this virtual server.

- *result*

  Address of a variable which receives the ID of the created virtual server.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. On success, the created virtual server will be automatically started.

Caution

You should *not* create a virtual server with an empty keypair except than the first time. If the server should crash, license problems might result when using “throw-away” keypairs, as the license systems might consider you are running more virtual servers than you actually do.

Instead query the keypair the first time the virtual server was started, save it to a file and reuse it when creating a new virtual server. This way licensing issues will not occur.

See the server sample which is included in the TeamSpeak 3 SDK for an example on how to save and restore keypairs.

Note

The TeamSpeak 3 server uses UDP. Support for TCP might be added in the future.

To query the keypair of a virtual server, use:

```
unsigned int ts3server_getVirtualServerKeyPair(serverID, result);

uint64 serverID;
char** result;
```

Parameters

- *serverID*

ID of the virtual server for which the keypair is queried.

- *result*

  Address of a variable that receives a string with the keypair of this virtual server. Save the keypair and pass it the next time this virtual server is created as parameter to ts3server_createVirtualServer.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If an error has occured, the result string is uninitialized and must not be released.

A virtual server can be stopped with:

```
unsigned int ts3server_stopVirtualServer(serverID);

uint64 serverID;
```

Parameters

- *serverID*

  ID of the virtual server that should be stopped.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Retrieve and store information

The Server Lib stores various pieces of information, which is made available to the custom server. This chapter covers how to query and store data in the Server Lib.

All strings passed to and from the Server Lib need to be encoded in UTF-8 format.

Client information

Query client information

Information about the clients currently connected to this virtual server can be retrieved and modified. To query client related information, use one of the following functions. The client is identified by the parameter *clientID*. The parameter *flag* is defined by the enum ClientProperties.

```
unsigned int ts3server_getClientVariableAsInt(serverID, clientID, flag, result);

uint64 serverID;
anyID clientID;
ClientProperties flag;
int* result;
```

```
unsigned int ts3server_getClientVariableAsString(serverID, clientID, flag, result);

uint64 serverID;
anyID clientID;
ClientProperties flag;
char** result;
```

Parameters

- *serverID*

  The ID of the virtual server on which the client property is queried.

- *clientID*

  ID of the client whose property is queried.

- *flag*

  Client propery to query, see below.

- *result*

  Address of a variable that receives the result value as int or string, depending on which function is used. In case of a string, memory must be released using `ts3server_freeMemory`, unless an error occured.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. For the string version: If an error has occured, the result string is uninitialized and must not be released.

The parameter *flag* specifies the type of queried information. It is defined by the enum ClientProperties:

```
enum ClientProperties {
 CLIENT_UNIQUE_IDENTIFIER = 0,    //automatically up-to-date for any client "in view", can be used
                                  //to identify this particular client installation
  CLIENT_NICKNAME,                //automatically up-to-date for any client "in view"
  CLIENT_VERSION,                 //for other clients than ourself, this needs to be requested
                                  //(=> requestClientVariables)
  CLIENT_PLATFORM,                //for other clients than ourself, this needs to be requested
                                  //(=> requestClientVariables)
  CLIENT_FLAG_TALKING,            //automatically up-to-date for any client that can be heard
                                  //(in room / whisper)
  CLIENT_INPUT_MUTED,             //automatically up-to-date for any client "in view", this clients
                                  //microphone mute status
  CLIENT_OUTPUT_MUTED,            //automatically up-to-date for any client "in view", this clients
                                  //headphones/speakers mute status
  CLIENT_OUTPUTONLY_MUTED         //automatically up-to-date for any client "in view", this clients
                                  //headphones/speakers only mute status
  CLIENT_INPUT_HARDWARE,          //automatically up-to-date for any client "in view", this clients
                                  //microphone hardware status (is the capture device opened?)
  CLIENT_OUTPUT_HARDWARE,         //automatically up-to-date for any client "in view", this clients
                                  //headphone/speakers hardware status (is the playback device opened?)
  CLIENT_INPUT_DEACTIVATED,       //only usable for ourself, not propagated to the network
  CLIENT_IDLE_TIME,               //internal use
  CLIENT_DEFAULT_CHANNEL,         //only usable for ourself, the default channel we used to connect
                                  //on our last connection attempt
  CLIENT_DEFAULT_CHANNEL_PASSWORD,//internal use
  CLIENT_SERVER_PASSWORD,         //internal use
  CLIENT_META_DATA,               //automatically up-to-date for any client "in view", not used by
```

```
                                //TeamSpeak, free storage for sdk users
  CLIENT_IS_MUTED,              //only make sense on the client side locally, "1" if this client is
                                //currently muted by us, "0" if he is not
  CLIENT_IS_RECORDING,          //automatically up-to-date for any client "in view"
  CLIENT_VOLUME_MODIFICATOR,    //internal use
  CLIENT_ENDMARKER,
};
```

- *CLIENT_UNIQUE_IDENTIFIER*

  String: Unique ID for this client. Stays the same after restarting the application, so you can use this to identify individual users.

- *CLIENT_NICKNAME*

  Nickname used by the client

- *CLIENT_VERSION*

  Application version used by this client.

- *CLIENT_PLATFORM*

  Operating system used by this client.

- *CLIENT_FLAG_TALKING*

  Set when the client is currently talking. Always available for visible clients.

- *CLIENT_INPUT_MUTED*

  Indicates the mute status of the clients capture device. Possible values are defined by the enum MuteInputStatus.

- *CLIENT_OUTPUT_MUTED*

  Indicates the combined mute status of the clients playback and capture devices. Possible values are defined by the enum MuteOutputStatus. Always available for visible clients.

- *CLIENT_OUTPUTONLY_MUTED*

  Indicates the mute status of the clients playback device. Possible values are defined by the enum MuteOutputStatus. Always available for visible clients.

- *CLIENT_INPUT_HARDWARE*

  Set if the clients capture device is not available. Possible values are defined by the enum HardwareInputStatus.

- *CLIENT_OUTPUT_HARDWARE*

  Set if the clients playback device is not available. Possible values are defined by the enum HardwareOutputStatus.

- *CLIENT_INPUT_DEACTIVATED*

  Set when the capture device has been deactivated as used in Push-To-Talk. Possible values are defined by the enum Input-DeactivationStatus. Only available to client, not propagated to the server.

- *CLIENT_IDLE_TIME*

Time the client has been idle.

- *CLIENT_TYPE*

  Indicates if the given client is a normal TeamSpeak 3 client or a connection established by the ServerQuery application.

- *CLIENT_DEFAULT_CHANNEL*

  CLIENT_DEFAULT_CHANNEL_PASSWORD

  Default channel name and password used in the last `ts3server_startConnection` call. Only available for own client.

- *CLIENT_META_DATA*

  Not used by TeamSpeak 3, offers free storage for SDK users.

- *CLIENT_IS_MUTED*

  Indicates a client has been locally muted with `ts3server_requestMuteClients`. Client-side only.

- *CLIENT_IS_RECORDING*

  Indicates a client is currently recording all voice data in his channel.

- *CLIENT_VOLUME_MODIFICATOR*

  The client volume modifier set by `ts3client_setClientVolumeModifier`.

Generally all types of information can be retrieved as both string or integer. However, in most cases the expected data type is obvious, like querying *CLIENT_NICKNAME* will clearly require to store the result as string.

Example: Query nickname of client with ID 123:

```
unsigned int error;
anyID clientID = 123;  /* Client ID in our example */
char* nickname;

if((error = ts3server_getClientVariableAsString(serverID, clientID, CLIENT_NICKNAME, &nickname)) != ERROR_ok) {
    printf("Error querying client nickname: %d\n", error);
    return;
}

printf("Client nickname is: %s\n", nickname);
ts3server_freeMemory(nickname);
```

Setting client information

Client information can be modified with

```
unsigned int ts3server_setClientVariableAsInt(serverID, clientID, flag, value);

uint64 serverID;
anyID clientID;
ClientProperties flag;
int value;
```

```
unsigned int ts3server_setClientVariableAsString(serverID, clientID, flag, value);

uint64 serverID;
anyID clientID;
ClientProperties flag;
const char* value;
```

Parameters

- *serverID*

  ID of the virtual server on which the client property should be changed.

- *clientID*

  ID of the client whose property should be changed.

- *flag*

  Client propery to query, see above.

- *value*

  Value the client property should be changed to.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Important

After modifying one or more client variables, you *must* flush the changes.

```
unsigned int ts3server_flushClientVariable(serverID, clientID);

uint64 serverID;
anyID clientID;
```

The idea behind flushing is, one can modify multiple values by calling ts3server_setClientVariableAsString and ts3server_setClientVariableAsInt and then apply all changes in one step.

For example, to change the nickname of the client with ID 55 to “Joe”:

```
anyID clientID = 55;  /* Client ID in our example */

/* Modifiy data */
if(ts3server_setClientVariableAsString(serverID, clientID, CLIENT_NICKNAME, "Joe") != ERROR_ok) {
    printf("Error setting client nickname\n");
    return;
}

/* Flush changes
if(ts3server_flushClientVariable(serverID, clientID) != ERROR_ok) {
    printf("Error flushing client variable\n");
```

```
}
```

Example for applying two changes:

```
anyID clientID = 66;  /* Client ID in our example */

/* Modify data 1 */
if(ts3server_setClientVariableAsInt(scHandlerID, clientID, CLIENT_AWAY, AWAY_ZZZ) != ERROR_ok) {
    printf("Error setting away mode\n");
    return;
}

/* Modify data 2 */
if(ts3server_setClientVariableAsString(scHandlerID, clientID, CLIENT_AWAY_MESSAGE, "Lunch") != ERROR_ok) {
    printf("Error setting away message\n");
    return;
}

/* Flush changes */
if(ts3server_flushClientVariable(scHandlerID, clientID) != ERROR_ok) {
    printf("Error flushing client variable");
}
```

Whisper lists

A client with a whisper list set can talk to the specified clients and channels. Whisper lists can be defined for individual clients. A whisper list consists of an array of client IDs and/or an array of channel IDs.

```
unsigned int ts3server_setClientWhisperList(serverID, clID, channelID, clientID);

uint64 serverID;
anyID clID;
const uint64* channelID;
const anyID* clientID;
```

Parameters

- *serverID*

  ID of the virtual server on which the whisper list is set.

- *clID*

  ID of the client whose whisper list is set.

- *channelID*

  NULL-terminated array of channel IDs. These channels will be added to the clients whisper list.

  Pass NULL for an empty list.

- *clientID*

  NULL-termianted array of client IDs. These clients will be added to the clients whisper list.

  Pass NULL for an empty list.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Channel information

Query channel information

Querying and modifying information related to channels is similar to dealing with clients. The parameter *flag* is defined by the enum ChannelProperties. The functions to query channel information are:

```
unsigned int ts3server_getChannelVariableAsInt(serverID, channelID, flag, result);

uint64 serverID;
uint64 channelID;
ChannelProperties flag;
int* result;


unsigned int ts3server_getChannelVariableAsString(serverID, channelID, flag, re-
sult);

uint64 serverID;
uint64 channelID;
ChannelProperties flag;
char** result;
```

Parameters

- *serverID*

  ID of the virtual server on which the channel property is queried.

- *channelID*

  ID of the queried channel.

- *flag*

  Channel propery to query, see below.

- *result*

  Address of a variable which receives the result value as int or string, depending on which function is used. In case of a string, memory must be released using ts3server_freeMemory, unless an error occured.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. For the string version: If an error has occured, the result string is uninitialized and must not be released.

The parameter *flag* specifies the type of queried information. It is defined by the enum ChannelProperties:

```
enum ChannelProperties {
  CHANNEL_NAME = 0,            //Available for all channels that are "in view", always up-to-date
```

```
  CHANNEL_TOPIC,                //Available for all channels that are "in view", always up-to-date
  CHANNEL_DESCRIPTION,          //Must be requested (=> requestChannelDescription)
  CHANNEL_PASSWORD,             //not available client side
  CHANNEL_CODEC,                //Available for all channels that are "in view", always up-to-date
  CHANNEL_CODEC_QUALITY,        //Available for all channels that are "in view", always up-to-date
  CHANNEL_MAXCLIENTS,           //Available for all channels that are "in view", always up-to-date
  CHANNEL_MAXFAMILYCLIENTS,     //Available for all channels that are "in view", always up-to-date
  CHANNEL_ORDER,                //Available for all channels that are "in view", always up-to-date
  CHANNEL_FLAG_PERMANENT,       //Available for all channels that are "in view", always up-to-date
  CHANNEL_FLAG_SEMI_PERMANENT,  //Available for all channels that are "in view", always up-to-date
  CHANNEL_FLAG_DEFAULT,         //Available for all channels that are "in view", always up-to-date
  CHANNEL_FLAG_PASSWORD,        //Available for all channels that are "in view", always up-to-date
  CHANNEL_CODEC_LATENCY_FACTOR, //Available for all channels that are "in view", always up-to-date
  CHANNEL_CODEC_IS_UNENCRYPTED, //Available for all channels that are "in view", always up-to-date
  CHANNEL_ENDMARKER,
};
```

- *CHANNEL_NAME*

  String: Name of the channel.

- *CHANNEL_TOPIC*

  String: Single-line channel topic.

- *CHANNEL_DESCRIPTION*

  String: Optional channel description. Can have multiple lines.

- *CHANNEL_PASSWORD*

  String: Password for password-protected channels.

  If a password is set or removed by modifying this field, *CHANNEL_FLAG_PASSWORD* will be automatically adjusted.

- *CHANNEL_CODEC*

  Int (0-3): Codec used for this channel:

  - 0 - Speex Narrowband (8 kHz)
  - 1 - Speex Wideband (16 kHz)
  - 2 - Speex Ultra-Wideband (32 kHz)

- *CHANNEL_CODEC_QUALITY*

  Int (0-10): Quality of channel codec of this channel. Valid values range from 0 to 10, default is 7. Higher values result in better speech quality but more bandwidth usage.

- *CHANNEL_MAXCLIENTS*

  Int: Number of maximum clients who can join this channel.

- *CHANNEL_MAXFAMILYCLIENTS*

  Int: Number of maximum clients who can join this channel and all subchannels.

- *CHANNEL_ORDER*

Int: Defines how channels are sorted in the GUI. Channel order is the ID of the predecessor channel after which this channel is to be sorted. If 0, the channel is sorted at the top of its hirarchy.

- *CHANNEL_FLAG_PERMANENT* / *CHANNEL_FLAG_SEMI_PERMANENT*

  Concerning channel durability, there are three types of channels:

  - Temporary

    Temporary channels have neither the *CHANNEL_FLAG_PERMANENT* nor *CHANNEL_FLAG_SEMI_PERMANENT* flag set. Temporary channels are automatically deleted by the server after the last user has left and the channel is empty. They will not be restored when the server restarts.

  - Semi-permanent

    Semi-permanent channels are not automatically deleted when the last user left but will not be restored when the server restarts.

  - Permanent

    Permanent channels will be restored when the server restarts.

- *CHANNEL_FLAG_DEFAULT*

  Int (0/1): Channel is the default channel. There can only be one default channel per server. New users who did not configure a channel to join on login in `ts3server_startConnection` will automatically join the default channel.

- *CHANNEL_FLAG_PASSWORD*

  Int (0/1): If set, channel is password protected. The password itself is stored in *CHANNEL_PASSWORD*.

- *CHANNEL_CODEC_LATENCY_FACTOR*

  (Int: 1-10): Latency of this channel. This allows to increase the packet size resulting in less bandwidth usage at the cost of higher latency. A value of 1 (default) is the best setting for lowest latency and best quality. If bandwidth or network quality are restricted, increasing the latency factor can help stabilize the connection. Higher latency values are only possible for low-quality codec and codec quality settings.

  For best voice quality a low latency factor is recommended.

- *CHANNEL_CODEC_IS_UNENCRYPTED*

  Int (0/1): If 1, this channel is not using encrypted voice data. If 0, voice data is encrypted for this channel. Note that channel voice data encryption can be globally disabled or enabled for the virtual server. Changing this flag makes only sense if global voice data encryption is set to be configured per channel as *CODEC_ENCRYPTION_PER_CHANNEL* (the default behaviour).

Example 1: Query topic of channel with ID 123:

```
uint64 channelID = 123;  /* Channel ID in our exampel */
char topic;

if(ts3server_getChannelVariableAsString(serverID, channel, CHANNEL_TOPIC, &topic) == ERROR_ok) {
    printf("Topic of channel %u is: %s\n", channelID, topic);
    ts3server_freeMemory(topic);
}
```

Setting channel information

Channel properties can be modified with:

```
unsigned int ts3server_setChannelVariableAsInt(serverID, channelID, flag, value);

uint64 serverID;
uint64 channelID;
ChannelProperties flag;
int value;


unsigned int ts3server_setChannelVariableAsString(serverID, channelID, flag, value);

uint64 serverID;
uint64 channelID;
ChannelProperties flag;
const char* value;
```

Parameters

- *serverConnectionHandlerID*

  ID of the virtual server on which the information for the specified channel should be changed.

- *channelID*

  ID of the channel whoses property should be changed.

- *flag*

  Channel propery to change, see above.

- *value*

  Value the channel property should be changed to.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Important

After modifying one or more channel variables, you *must* flush the changes.

```
unsigned int ts3server_flushChannelVariable(serverID, channelID);

uint64 serverID;
uint64 channelID;
```

Example: Change the channel name and topic:

```
/* Modify channel name */
if(ts3server_setChannelVariableAsString(serverID, channelID, CHANNEL_NAME,  "New channel name") != ERROR_ok) {
```

```
    printf("Error setting channel name\n");
}

/* Modify channel topic */
if(ts3server_setChannelVariableAsString(serverID, channelID, CHANNEL_TOPIC, "New channel topic") != ERROR_ok) {
    printf("Error setting channel topic\n");
}

/* Flush changes */
if(ts3server_flushChannelVariable(serverID, channelID) != ERROR_ok) {
    printf("Error flushing channel variable\n");
}
```

Server information

Query server information

Information related to a virtual server can be queried with::

```
unsigned int ts3server_getVirtualServerVariableAsInt(serverID, flag, result);

uint64 serverID;
VirtualServerProperties flag;
int* result;


unsigned int ts3server_getVirtualServerVariableAsString(serverID, flag, result);

uint64 serverID;
VirtualServerProperties flag;
char** result;
```

Parameters

- *serverID*

  ID of the virtual server of which the property is queried.

- *flag*

  Virtual server propery to query, see below.

- *result*

  Address of a variable which receives the result value as int or string, depending on which function is used. In case of a string, memory must be released using `ts3server_freeMemory`, unless an error occured.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. For the string version: If an error has occured, the result string is uninitialized and must not be released.

The parameter *flag* specifies the type of queried information. It is defined by the enum VirtualServerProperties:

```
enum VirtualServerProperties {
  VIRTUALSERVER_UNIQUE_IDENTIFIER = 0, //available when connected, can be used to identify this particular
```

```
                                    //server installation
  VIRTUALSERVER_NAME,               //available and always up-to-date when connected
  VIRTUALSERVER_WELCOMEMESSAGE,     //available when connected, not updated while connected
  VIRTUALSERVER_PLATFORM,           //available when connected
  VIRTUALSERVER_VERSION,            //available when connected
  VIRTUALSERVER_MAXCLIENTS,         //only available on request (=> requestServerVariables), stores the
                                    //maximum number of clients that may currently join the server
  VIRTUALSERVER_PASSWORD,           //not available to clients, the server password
  VIRTUALSERVER_CLIENTS_ONLINE,     //only available on request (=> requestServerVariables),
  VIRTUALSERVER_CHANNELS_ONLINE,    //only available on request (=> requestServerVariables),
  VIRTUALSERVER_CREATED,            //available when connected, stores the time when the server was created
  VIRTUALSERVER_UPTIME,             //only available on request (=> requestServerVariables), the time
                                    //since the server was started
  VIRTUALSERVER_CODEC_ENCRYPTION_MODE, //available and always up-to-date when connected
  VIRTUALSERVER_ENDMARKER,
};
```

- *VIRTUALSERVER_UNIQUE_IDENTIFIER*

  Unique ID for this virtual server. Stays the same after restarting the server application.

- *VIRTUALSERVER_NAME*

  Name of this virtual server.

- *VIRTUALSERVER_WELCOMEMESSAGE*

  Optional welcome message sent to the client on login.

- *VIRTUALSERVER_PLATFORM*

  Operating system used by this server.

- *VIRTUALSERVER_VERSION*

  Application version of this server.

- *VIRTUALSERVER_MAXCLIENTS*

  Defines maximum number of clients which may connect to this server.

- *VIRTUALSERVER_PASSWORD*

  Optional password of this server.

  If a password is set or removed by modifying this field, *VIRTUALSERVER_FLAG_PASSWORD* will be automatically adjusted.

- *VIRTUALSERVER_CLIENTS_ONLINE*

  VIRTUALSERVER_CHANNELS_ONLINE

  Number of clients and channels currently on this virtual server.

- *VIRTUALSERVER_CREATED*

  Time when this virtual server was created.

- *VIRTUALSERVER_UPTIME*

Uptime of this virtual server.

- *VIRTUALSERVER_CODEC_ENCRYPTION_MODE*

  Defines if voice data encryption is configured per channel, globally forced on or globally forced off for this virtual server. The default behaviour is configure per channel, in this case modifying the channel property *CHANNEL_CODEC_IS_UNENCRYPTED* defines voice data encryption of individual channels.

  Virtual server encryption mode can be set to the following parameters:

```
enum CodecEncryptionMode {
   CODEC_ENCRYPTION_PER_CHANNEL = 0,
   CODEC_ENCRYPTION_FORCED_OFF,
   CODEC_ENCRYPTION_FORCED_ON,
};
```

  This property is always available when connected.

Example checking the number of clients online, obviously an integer value:

```
int clientsOnline;

if(ts3server_getVirtualServerVariableAsInt(serverID, VIRTUALSERVER_CLIENTS_ONLINE,
                                           &clientsOnline) == ERROR_ok)
    printf("There are %d clients online\n", clientsOnline);
```

Setting server information

Change server variables with the following functions:

```
unsigned int ts3server_setVirtualServerVariableAsInt(serverID, flag, value);

uint64 serverID;
ChannelProperties flag;
int value;


unsigned int ts3server_setVirtualServerVariableAsString(serverID, flag, value);

uint64 serverID;
ChannelProperties flag;
const char* value;
```

Parameters

- *serverID*

  ID of the virtual server of which the property should be changed.

- *flag*

  Virtual server propery to change, see above.

- *value*

Value the virtual server property should be changed to.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Important

After modifying one or more server variables, you *must* flush the changes.

```
unsigned int ts3server_flushVirtualServerVariable(serverID);

uint64 serverID;
```

Example: Change the servers welcome message:

```
if(ts3server_setVirtualServerVariableAsString(serverID, VIRTUALSERVER_WELCOMEMESSAGE,
                                              "New welcome message") != ERROR_ok) {
    printf("Error setting server welcomemessage\n");
    return;
}

if(ts3server_flushVirtualServerVariable(serverID) != ERROR_ok) {
    printf("Error flushing server variable\n");
}
```

Bandwidth information

The server offers information about the currently used bandwidth.

The following set of connection properties can be queried:

- CONNECTION_PACKETS_SENT_TOTAL
- CONNECTION_BYTES_SENT_TOTAL
- CONNECTION_PACKETS_RECEIVED_TOTAL
- CONNECTION_BYTES_RECEIVED_TOTAL
- CONNECTION_BANDWIDTH_SENT_LAST_SECOND_TOTAL
- CONNECTION_BANDWIDTH_SENT_LAST_MINUTE_TOTAL
- CONNECTION_BANDWIDTH_RECEIVED_LAST_SECOND_TOTAL
- CONNECTION_BANDWIDTH_RECEIVED_LAST_MINUTE_TOTAL

The connection information can be queried with the following two functions:

```
unsigned int ts3server_getVirtualServerConnectionVariableAsUInt64(serverID, flag,
result);

uint64 serverID;
enum ConnectionProperties flag;
```

```
uint64* result;
```

```
unsigned int ts3server_getVirtualServerConnectionVariableAsDouble(serverID, flag,
result);

uint64 serverID;
enum ConnectionProperties flag;
double* result;
```

Parameters

- *serverID*

  Server ID

- *flag*

  One of the above listed connection properties.

- *result*

  Address of a variable that receives the result value as uint64 (unsigned 64-bit integer) or double type, depending on which of the two functions was used.

Both functions return *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Channel and client manipulation

The Server Lib offers a subset of client-side functionality to create, move and delete channels directly on the server.

Creating a new channel

To create a channel, first set the desired channel variables using `ts3server_setChannelVariableAsInt` and `ts3server_setChannelVariableAsString`. Pass zero as the channel ID parameter.

Next send the request to the server by calling:

```
unsigned int ts3server_flushChannelCreation(serverID, channelParentID, result);

uint64 serverID;
uint64 channelParentID;
uint64* result;
```

Parameters

- *serverID*

ID of the virtual server on which that channel should be created.

- *channelParentID*

  ID of the parent channel, if the new channel is to be created as subchannel. Pass zero if the channel should be created as top-level channel.

- *result*

  Address of a variable that receives the ID of the newly created channel.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Example code to create a channel:

```
#define CHECK_ERROR(x) if((error = x) != ERROR_ok) { goto on_error; }

int createChannel(uint64 serverID, uint64 parentChannelID, const char* name, const char* topic,
                  const char* description, const char* password, int codec, int codecQuality,
                  int maxClients, int familyMaxClients, int order, int perm, int semiperm,
                  int default) {
  unsigned int error;
  uint64 newChannelID;

  /* Set channel data, pass 0 as channel ID */
  CHECK_ERROR(ts3server_setChannelVariableAsString(serverID, 0, CHANNEL_NAME, name));
  CHECK_ERROR(ts3server_setChannelVariableAsString(serverID, 0, CHANNEL_TOPIC, topic));
  CHECK_ERROR(ts3server_setChannelVariableAsString(serverID, 0, CHANNEL_DESCRIPTION, description));
  CHECK_ERROR(ts3server_setChannelVariableAsString(serverID, 0, CHANNEL_PASSWORD, password));
  CHECK_ERROR(ts3server_setChannelVariableAsInt   (serverID, 0, CHANNEL_CODEC, codec));
  CHECK_ERROR(ts3server_setChannelVariableAsInt   (serverID, 0, CHANNEL_CODEC_QUALITY, codecQuality));
  CHECK_ERROR(ts3server_setChannelVariableAsInt   (serverID, 0, CHANNEL_MAXCLIENTS, maxClients));
  CHECK_ERROR(ts3server_setChannelVariableAsInt   (serverID, 0, CHANNEL_MAXFAMILYCLIENTS, familyMaxClients));
  CHECK_ERROR(ts3server_setChannelVariableAsInt   (serverID, 0, CHANNEL_ORDER, order));
  CHECK_ERROR(ts3server_setChannelVariableAsInt   (serverID, 0, CHANNEL_FLAG_PERMANENT, perm));
  CHECK_ERROR(ts3server_setChannelVariableAsInt   (serverID, 0, CHANNEL_FLAG_SEMI_PERMANENT, semiperm));
  CHECK_ERROR(ts3server_setChannelVariableAsInt   (serverID, 0, CHANNEL_FLAG_DEFAULT, default));

  /* Flush changes to server */
  CHECK_ERROR(ts3server_flushChannelCreation(serverID, parentChannelID, &newChannelID));

  printf("Created new channel with ID: %u\n", newChannelID);
  return 0;  /* Success */

on_error:
  printf("Error creating channel: %d\n", error);
  return 1;  /* Failure */
}
```

After creating a channel, the event onChannelCreated is called.

Deleting a channel

A channel can be deleted by the server with

```
unsigned int ts3server_channelDelete(serverID, channelID, force);

uint64 serverID;
uint64 channelID;
```

```
int force;
```

Parameters

- *serverID*

  The ID of the virtual server on which the channel should be deleted.

- *channelID*

  The ID of the channel to be deleted.

- *force*

  If 1, first move all clients inside the specified channel to the default channel and then delete the specific channel. If false, deleting a channel with joined clients will fail.

  If 0, the server will refuse to a channel that is not empty.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

After successfully deleting a channel, the event onChannelDeleted is called for every deleted channel.

Moving a channel

To move a channel to a new parent channel, call

```
unsigned int ts3server_channelMove(serverID, channelID, newChannelParentID);
```

```
uint64 serverID;
uint64 channelID;
uint64 newChannelParentID;
```

Parameters

- *serverID*

  ID of the virtual server on which the channel should be moved.

- *channelID*

  ID of the channel to be moved.

- *newChannelParentID*

  ID of the parent channel where the moved channel is to be inserted as child. Use *0* to insert as top-level channel.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

After the channel has been moved, the event onChannelEdited is called.

Moving clients

Clients can be moved server-side to another channel, in addition to the client-side functionality offered by the Client Lib. To move one or multiple clients to a new channel, call:

```
unsigned int ts3server_clientMove(serverID, newChannelID, clientIDArray);

uint64 serverID;
uint64 newChannelID;
const anyID* clientIDArray;
```

Parameters

- *serverID*

  ID of the virtual server on which the client should be moved.

- *newChannelID*

  ID of the channel in which the clients should be moved into.

- *newChannelParentID*

  Zero-terminated array with the IDs of the clients to be moved.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

After the channel has been moved, the event `onClientMoved` is called.

Example to move a single client to another channel:

```
anyID clientIDArray[2];  /* One client plus terminating zero as end-marker */
uint64 newChannelID;
unsigned int error;

clientIDArray[0] = clientID;  /* Client to move */
clientIDArray[1] = 0;  /* End marker */

if((error = ts3server_clientMove(serverID, newChannelID, channelIDArray)) != ERROR_ok) {
    /* Handle error */
    return;
}

/* Client moved successfully */
```

Events

The server lib will notify the server application about certain actions by sending events as callbacks. Callback function pointers needs to be initialized in `ts3server_initServerLib`.

Note

Your callback implementations should exit quickly to avoid blocking the server. If you require to do lengthly operations, consider using a new thread to let the callback itself finish as soon as possible.

All strings are UTF-8 encoded.

A client has connected:

```
void onClientConnected(serverID, clientID, channelID, removeClientError);

uint64 serverID;
anyID clientID;
uint64 channelID;
unsigned int* removeClientError;
```

Parameters

- *serverID*

  ID of the virtual server.

- *clientID*

  ID of the connected client.

- *channelID*

  ID of the channel the client has joined.

- *removeClientError*

  If the pointer value is *ERROR_ok* (default), this client will connect normally to the virtual server. To prevent the client connecting, set the pointer value to any valid error (see the header `public_errors.h`):

  ```
  *removeClientError = ERROR_client_insufficient_permissions;
  ```

  If you do not want to block the client, it's best to not modify the *removeClientError* parameter at all and leave the default value of *ERROR_ok*.

A client has disconnected:

```
void onClientDisconnected(serverID, clientID, channelID);

uint64 serverID;
anyID clientID;
uint64 channelID;
```

Parameters

- *serverID*

  ID of the virtual server.

- *clientID*

  ID of the disconnected client.

- *channelID*

  ID of the channel the client has left.

A client has moved into another channel:

```
void onClientMoved(serverID, clientID, oldChannelID, newChannelID);

uint64 serverID;
anyID clientID;
uint64 oldChannelID;
uint64 newChannelID;
```

Parameters

- *serverID*

  ID of the virtual server.

- *clientID*

  ID of the moved client.

- *oldChannelID*

  ID of the old channel the client has left.

- *newChannelID*

  ID of the new channel the client has joined.

A channel has been created:

```
void onChannelCreated(serverID, invokerClientID, channelID);

uint64 serverID;
anyID invokerClientID;
uint64 channelID;
```

Parameters

- *serverID*

  ID of the virtual server.

- *invokerClientID*

  ID of the invoker who created the channel (client or server ID).

- *channelID*

  ID of the created channel.

A channel has been edited:

```
void onChannelEdited(serverID, invokerClientID, channelID);

uint64 serverID;
anyID invokerClientID;
uint64 channelID;
```

Parameters

- *serverID*

  ID of the virtual server.

- *invokerClientID*

  ID of the invoker who edited the channel (client or server ID).

- *channelID*

  ID of the edited channel.

A channel has been deleted:

```
void onChannelDeleted(serverID, invokerClientID, channelID);

uint64 serverID;
anyID invokerClientID;
uint64 channelID;
```

Parameters

- *serverID*

  ID of the virtual server.

- *invokerClientID*

  ID of the invoker who deleted the channel (client or server ID).

- *channelID*

  ID of the deleted channel.

Text messages can be received on the server side. Only server and channel chats trigger this event, client-to-client messages are not caught for privacy reasons.

Server chat messages can be intercepted with:

```
void onServerTextMessageEvent(serverID, invokerClientID, textMessage);

uint64 serverID;
anyID invokerClientID;
const char* textMessage;
```

Parameters

- *serverID*

  ID of the virtual server.

- *invokerClientID*

  ID of the client who sent the text message.

- *textMessage*

  Message text

Channel chat messages can be intercepted with:

```
void onChannelTextMessageEvent(serverID, invokerClientID, targetChannelID, textMes-
sage);

uint64 serverID;
anyID invokerClientID;
uint64 targetChannelID;
const char* textMessage;
```

Parameters

- *serverID*

  ID of the virtual server.

- *invokerClientID*

  ID of the client who sent the text message.

- *targetChannelID*

  ID of the channel in which the text message was sent.

- *textMessage*

  Message text

If user-defined logging was enabled when initialzing the Server Lib by passing *LogType_USERLOGGING* to the *usedLogTypes* parameter of ts3server_initServerLib, log messages will be sent to the following callback, which allows user customizable logging and handling or critical errors:

```
void onUserLoggingMessageEvent(logMessage, logLevel, logChannel, logID, logTime, completeLogString);

const char* logMessage;
int logLevel;
const char* logChannel;
uint64 logID;
const char* logTime;
const char* completeLogString;
```

Parameters

- *logMessage*

  Actual log message text.

- *logLevel*

  Severity of log message, defined by the enum LogLevel.

  ```
  enum LogLevel {
      LogLevel_CRITICAL = 0, //these messages stop the program
      LogLevel_ERROR,        //everything that is really bad, but not so bad we need to shut down
      LogLevel_WARNING,      //everything that *might* be bad
      LogLevel_DEBUG,        //output that might help find a problem
      LogLevel_INFO,         //informational output, like "starting database version x.y.z"
      LogLevel_DEVEL         //developer only output (will not be displayed in release mode)
  };
  ```

  Note that only log messages of a level higher than the one configured with ts3server_setLogVerbosity will appear.

- *logChannel*

  Optional custom text to categorize the message channel.

- *logID*

  Virtual server ID identifying the current virtual server when using multiple connections.

- *logTime*

  String with date and time when the log message occured.

- *completeLogString*

  Provides a verbose log message including all previous parameters for convinience.

A client connected to this server starts or stops talking:

```
void onClientStartTalkingEvent(serverID, clientID);
```

```
uint64 serverID;
anyID clientID;
```

```
void onClientStopTalkingEvent(serverID, clientID);
```

```
uint64 serverID;
anyID clientID;
```

Parameters

- *serverID*

  The ID of the server which sent the event.

- *clientID*

  ID of the client who starts or stops talking

If required, the raw voice data can be caught by the server to implement server-side voice recording. Whenever a client is sending voice data, the following function is called:

```
void onVoiceDataEvent(serverID, clientID, voiceData, voiceDataSize, frequency);
```

```
uint64 serverID;
anyID clientID;
unsigned char* voiceData;
unsigned int voiceDataSize;
unsigned int frequency;
```

Parameters

- *serverID*

  The ID of the server which sent the event.

- *clientID*

  ID of the client who sent the voice data.

- *voiceData*

  Buffer containing the voice data. Format is 16 bit mono.

Caution

The buffer must not be freed.

- *voiceDataSize*

Size of the *voiceData* buffer.

- *frequency*

  Frequency of the voice data.

Note

This event is always fired, even if the client is the only user in a channel. So clients “talking to themselves” will also be recorded.

If server-side recording is not required, don't implement this callback.

The following event is called when a license error occurs, like for example missing license file, expired license, starting too many virtual servers etc. Instead of shutting down the whole process by throwing a critical error in the Server Lib, this callback allows you to handle the issue gracefully and keep your application running.

```
void onAccountingErrorEvent(serverID, errorCode);
```

```
uint64 serverID;
unsigned int errorCode;
```

Parameters

- *serverID*

  The ID of the virtual server on which the license error occured. This virtual server will be automatically shutdown, other virtual servers keep running.

  If *serverID* is zero, all virtual servers are affected and have been shutdown. In this case you might want to call `ts3server_destroyServerLib` to clean up resources.

- *errorCode*

  Code of the occured error. Use `ts3server_getGlobalErrorMessage` to convert to a message string.

Custom encryption

As an optional feature, the TeamSpeak 3 SDK allows users to implement custom encryption and decryption for all network traffic. Custom encryption replaces the default AES encryption implemented by the TeamSpeak 3 SDK. A possible reason to apply own encryption might be to make ones TeamSpeak 3 client/server incompatible to other SDK implementations.

Custom encryption must be implemented the same way in both the client and server.

Note

If you do not want to use this feature, just don't implement the two encryption callbacks.

To encrypt outgoing data, implement the callback:

```
void onCustomPacketEncryptEvent(dataToSend, sizeOfData);

char** dataToSend;
unsigned int* sizeOfData;
```

Parameters

- *dataToSend*

  Pointer to an array with the outgoing data to be encrypted.

  Apply your custom encryption to the data array. If the encrypted data is smaller than sizeOfData, write your encrypted data into the existing memory of dataToSend. If your encrypted data is larger, you need to allocate memory and redirect the pointer dataToSend. You need to take care of freeing your own allocated memory yourself. The memory allocated by the SDK, to which dataToSend is originally pointing to, must not be freed.

- *sizeOfData*

  Pointer to an integer value containing the size of the data array.

To decrypt incoming data, implement the callback:

```
void onCustomPacketDecryptEvent(dataReceived, dataReceivedSize);

char** dataReceived;
unsigned int* dataReceivedSize;
```

Parameters

- *dataReceived*

  Pointer to an array with the received data to be decrypted.

  Apply your custom decryption to the data array. If the decrypted data is smaller than dataReceivedSize, write your decrypted data into the existing memory of dataReceived. If your decrypted data is larger, you need to allocate memory and redirect the pointer dataReceived. You need to take care of freeing your own allocated memory yourself. The memory allocated by the SDK, to which dataReceived is originally pointing to, must not be freed.

- *dataReceivedSize*

  Pointer to an integer value containing the size of the data array.

Example code implementing a very simple XOR custom encryption and decryption (also see the SDK examples):

```
void onCustomPacketEncryptEvent(char** dataToSend, unsigned int* sizeOfData) {
    unsigned int i;
    for(i = 0; i < *sizeOfData; i++) {
        (*dataToSend)[i] ^= CUSTOM_CRYPT_KEY;
    }
}
```

```
void onCustomPacketDecryptEvent(char** dataReceived, unsigned int* dataReceivedSize) {
    unsigned int i;
    for(i = 0; i < *dataReceivedSize; i++) {
        (*dataReceived)[i] ^= CUSTOM_CRYPT_KEY;
    }
}
```

Miscellaneous functions

Memory dynamically allocated in the Server Lib needs to be released with:

```
unsigned int ts3server_freeMemory(pointer);
```

```
void* pointer;
```

Parameters

- *pointer*

  Address of the variable to be released.

Example:

```
char* version;

if(ts3server_getServerLibVersion(&version) == ERROR_ok) {
    printf("Version: %s\n", version);
    ts3server_freeMemory(version);
}
```

Important

Memory must not be released if the function, which dynamically allocated the memory, returned an error. In that case, the result is undefined and not initialized, so freeing the memory might crash the application.

The severity of log messages that are passed to the callback `onUserLoggingMessageEvent` can be configured with:

```
unsigned int ts3server_setLogVerbosity(logVerbosity);
```

```
enum LogLevel logVerbosity;
```

Parameters

- *logVerbosity*

  Only messages with a LogLevel equal or higher than *logVerbosity* will be sent to the callback.

  The default value is *LogLevel_DEVEL*.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

For example, after calling

```
ts3server_setLogVerbosity(LogLevel_ERROR);
```

only log messages of level *LogLevel_ERROR* and *LogLevel_CRITICAL* will be passed to `onUserLoggingMessageEvent`.

FAQ

- I cannot start multiple server processes? I cannot start more than one virtual server?
- How can I configure the maximum number of slots?
- I get "Accounting | | sid=1 is running" "initializing shutdown" in the log
- How to implement a name/password authentication?

I cannot start multiple server processes? I cannot start more than one virtual server?

You don't have a valid license key in the correct location. The file `licensekey.dat` needs to be placed in the same directory as your server executable. If no or an invalid license key is present, the server will run with the following restrictions:

- Only one server process per machine
- Only one virtual server per process
- Only 32 slots

Please contact <sales@tritoncia.com> about license key inquiries or to obtain a valid license.

How can I configure the maximum number of slots?

The number of slots per virtual server can be changed by setting the virtual server property *VIRTUALSERVER_MAXCLIENTS*.

Example to set 100 slots on the specified virtual server:

```
ts3server_setVirtualServerVariableAsInt(serverID, VIRTUALSERVER_MAXCLIENTS, 100);  // Set value
ts3server_flushVirtualServerVariable(serverID);  // Flush value
```

Important

Please note that you probably do not have unlimited slots allowed by your license, so don't set this arbitrarily.

I get "Accounting | | sid=1 is running" "initializing shutdown" in the log

This error does not occur because you are exceeding your licensed server or slot count, but rather because you are running more than one instance of a virtual server with the same server keypair.

When creating a new virtual server, a keypair must be passed to `ts3server_createVirtualServer`. It is important to store the used keypair and reuse it when restarting this virtual server later instead of creating a new key. See the server sample within the SDK for an example.

However, above problem can happen if the virtual server is started with a stored keypair, then the entire folder including the stored keypair is copied to another PC and also started there with the *same* key. In this case the licensing server will notice the same key is used more than once after one hour and shutdown the most recently started server which tried to steal the identity of an already running server.

The fix, in the server sample case, would be to delete the keypair_*.txt files from the copied directory before starting the second server, that way a new key would be generated and the licensing server would see the two servers as two valid different entities. The accounting server would now only complain if the number of simultaneously running servers exceeds your number of slots.

How to implement a name/password authentication?

Although TeamSpeak 3 offers an authentication system based on public/private keys, an often made request is to use an additional login name/password mechanism to authenticate clients with the TeamSpeak 3 server. Here we will suggest a possibility to implement this authentication on top of the existing public/private key mechanism.

When connecting to the TeamSpeak 3 server, a client might make use of the CLIENT_META_DATA property and fill this with a name/password combination to let the server validate this this data in the servers `onClientConnected` callback. This callback allows to set an error value to block this clients connection.

The client-side code:

```
// In the client, set CLIENT_META_DATA before connecting
if(ts3client_setClientSelfVariableAsString(scHandlerID, CLIENT_META_DATA, "NAME#PASSWORD") != ERROR_ok) {
    printf("Failed setting client meta data\n");
    return;
}

// Call ts3client_startConnection
```

In the server implement the onClientConnected callback, which validates the name/password meta data and refuses the connection if not validated:

```
void onClientConnected(uint64 serverID, anyID clientID, uint64 channelID, unsigned int* removeClientError) {
    // Query CLIENT_META_DATA
    char* metaData;
    if(ts3server_getClientVariableAsString(serverID, clientID, CLIENT_META_DATA, &metaData) != ERROR_ok) {
        printf("Failed querying client meta data\n");
        *removeClientError = ERROR_client_not_logged_in;  // Block client
        return;
    }

    // Validate name/password
    if(!validateNamePassword(metaData)) {
        *removeClientError = ERROR_client_not_logged_in;  // Block client
    }
    // Client is allowed to connect if removeClientError is not changed
    // (defaults is ERROR_ok)
    ts3server_freeMemory(metaData);  // Release previously allocated memory
}
```

Index

L

M

S

T

W