TeamSpeak 3 Server SDK Developer Manual

Revision 2017-09-08 09:51:42

Table of Contents

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 XP, Vista, 7, 8, 8.1 (32- and 64-bit)

- Mac OS X

  Mac OS X 10.6 and above

- Linux

  Any recent Linux distribution with libstdc++ 6 (32- and 64-bit)

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@teamspeakusa.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.

- *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*

  If file logging is used, this defines the location where the logs are written 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`.

Note

During initialization the serverlib will attempt to connect to the TeamSpeak licensing server. This function may block if the licensing server is unreachable.

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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *`serverPort`*

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

- *`serverIp`*

  Comma seperated list of IP addresses to bind the virtual server to. Both IPv4 and IPv6 IPs are supported. 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.

Caution

When a virtual server is started, it will register itself at a TeamSpeak licensing server reporting the maximal client count to ensure the server is operating within the license limits. On shutdown, the virtual server will unregister at the licensing server.

This leads to two important things to keep in mind:

1) Don't just kill your server with Ctrl-C, instead ensure it's shutdown properly calling `ts3server_stopVirtualServer`. If killed too often, the licensing server might reject this server instance because the license seems to be exceeded as client slots are only added but never removed.

2) Don't start a virtual server too frequently. This may raise an error *“virtualserver started too many times in a certain time period”*. Contacting the licensing server will be prevented to protect our backend services from getting spammed with frequent server updates for the same server.

The trigger conditions for this error are very specific and are as follows:

- You start a server instance and then stop the same instance within 5 seconds of it being started.
- The above happens more than 3 times in a 41 minute window.

After you triggered this error, you will be prevented from starting any server instance using the affected license for a period of 12 minutes.

You should first investigate how you managed to trigger this error and change your scripts to avoid triggering the conditions above. Then you must not start any server instance for the 12 minute grace period mentioned above. After this wait, you should be able to start your instance normally.

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;
```

- *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;
```

- *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`.

Alternative way to create virtual servers

In addition to the previously mentioned way to start a virtual server using `ts3server_startVirtualServer`, there is an alternative way to create a virtual server. The advantage of this method is the possibility to restore the complete channel structure on server creation including the channel IDs in one step. This allows taking and restoring server snapshots including the channel tree.

Note

For a complete example please see the “server_creation_params” sample code in the SDK package.

First, a struct *TS3VirtualServerCreationParams* has to be created, which will be filled with essential and optional server parameters and then used to create and start the virtual server. Create a new virtual server parameter structure with the function:

```
unsigned int ts3server_makeVirtualServerCreationParams(result);
```

```
struct TS3VirtualServerCreationParams** result;
```

- *result*

  Address of a variable that receives the created struct *TS3VirtualServerCreationParams*.

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

Once the struct *TS3VirtualServerCreationParams* has been created, it needs to be filled with the essential parameters to create a new virtual server. Essential parameters include server port, IP, the key pair, max clients, number of channels we want to start the server with and the virtual server ID.

Set the essential parameters with the function:

```
unsigned int ts3server_setVirtualServerCreationParams(virtualServerCreationParams,
serverPort, serverIp, serverKeyPair, serverMaxClients, channelCount, serverID);

struct TS3VirtualServerCreationParams* virtualServerCreationParams;
unsigned int serverPort;
const char* serverIp;
const char* serverKeyPair;
unsigned int serverMaxClients;
unsigned int channelCount;
uint64 serverID;
```

- *virtualServerCreationParams*

  Address of the struct *TS3VirtualServerCreationParams*, which was created earlier with ts3server_makeVirtualServerCreationParams.

  serverPort

  Port of the virtual server to be created.

  serverIp

  Comma seperated list of IP addresses to bind the virtual server to. Both IPv4 and IPv6 IPs are supported. Pass “0.0.0.0, ::” to bind the virtual server to all IP addresses.

  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 numbers clients which can be online on the virtual server at the same time.

channelCount

Number of channels which are immediately created at server start. This value defines how many structs *TS3ChannelCreationParams* are available in the function ts3server_getVirtualServerCreationParamsChannelCreationParams, where the channels will be defined later.

serverID

ID of the virtual server to be created.

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

After essential virtual server parameters have been defined with ts3server_setVirtualServerCreationParams, additional parameters can be filled. For that, first a struct *TS3Variables* needs to be queried from the *TS3VirtualServerCreationParams*, in which the additional paramters will be written using the functions ts3server_setVariableAsInt, ts3server_setVariableAsUInt64 and ts3server_setVariableAsString.

Query the *TS3Variables* with:

```
unsigned int
ts3server_getVirtualServerCreationParamsVariables(virtualServerCreationParams, result);

struct TS3VirtualServerCreationParams* virtualServerCreationParams;
struct TS3Variables** result;
```

- *virtualServerCreationParams*

  Address of the struct *TS3VirtualServerCreationParams*, which was created earlier with ts3server_makeVirtualServerCreationParams.

  result

  Adress of a variable into which the struct *TS3Variables* is written.

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

Once you have a pointer to a valid struct *TS3Variables*, you can query and modify various parameters with one of the following functions. Select the proper function depending of the type of the parameter you want to query or modify. The parameter *flag* is a value *VIRTUALSERVER_** defined in the enum *VirtualServerProperties*.

Query parameter of the struct *TS3Variables* as integer:

```
unsigned int ts3server_getVariableAsInt(var, flag, result);

struct TS3Variables* var;
int flag;
```

```
int* result;
```

Query parameter of the struct *TS3Variables* as uint64:

```
unsigned int ts3server_getVariableAsUInt64(var, flag, result);

struct TS3Variables* var;
int flag;
uint64* result;
```

Query parameter of the struct *TS3Variables* as string:

```
unsigned int ts3server_getVariableAsString(var, flag, result);

struct TS3Variables* var;
int flag;
char** result;
```

Set parameter of the struct *TS3Variables* as integer:

```
unsigned int ts3server_setVariableAsInt(var, flag, value);

struct TS3Variables* var;
int flag;
int value;
```

Set parameter of the struct *TS3Variables* as uint64:

```
unsigned int ts3server_setVariableAsUInt64(var, flag, value);

struct TS3Variables* var;
int flag;
uint64 value;
```

Set parameter of the struct *TS3Variables* as string:

```
unsigned int ts3server_setVariableAsString(var, flag, value);

struct TS3Variables* var;
int flag;
const char* value;
```

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

Example setting the virtual server name:

```
if(ts3server_setVariableAsString(vars, VIRTUALSERVER_NAME, "My Server") != ERROR_ok) {
    printf("Failed to set virtual server name: %d\n", error);
}
```

After setting global virtual server parameters we are ready to initialize the channel tree. In a loop, for each channel you retrieve a struct *TS3ChannelCreationParams* and fill it with the desired channel parameters, including the channel ID.

Note

The number of created channels must match exactly the previously defined *channelCount* in ts3server_setVirtualServerCreationParams.

For each channel get the channel creation param for the given channel index. This channel param structs are subobjects created inside the server creation params, so do not delete them.

unsigned int **ts3server_getVirtualServerCreationParamsChannelCreationParams**(*virtualServerCreationParams*, *channelIdx*, *result*);

```
struct TS3VirtualServerCreationParams* virtualServerCreationParams;
unsigned int channelIdx;
struct TS3ChannelCreationParams** result;
```

- *virtualServerCreationParams*

  Address of the struct *TS3VirtualServerCreationParams*, which was created earlier with ts3server_makeVirtualServerCreationParams.

  channelIdx

  Index of the channel we want to address. Index must be in range of the previously specified *channelCount* of this virtual server.

  result

  Address of a struct *TS3ChannelCreationParams*, which we are going to fill in the next step.

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

Once we have a struct *TS3ChannelCreationParams* for this channel, we can start to fill it in two steps. Step 1 is setting the essential data, step 2 is setting optional additional data.

Essential parameters are channel parent ID and channel ID. Set them with

unsigned int **ts3server_setChannelCreationParams**(*channelCreationParams*, *channelParentID*, *channelID*);

```
struct TS3ChannelCreationParams* channelCreationParams;
uint64 channelParentID;
uint64 channelID;
```

- *channelCreationParams*

  Address of the struct *TS3ChannelCreationParams*, where the parameters should be set.

  channelParentID

  ID of the parent channel. Set 0 to create this as top-level channel.

  channelID

  ID of the channel to create. This allows you to setup a complete channel tree with predefined channel IDs, unlike the legacy way to create channels where the channel ID is automatically assigned by the server.

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

After setting the essential parameters, we can set optional additional parameters in a way similar as above for the virtual server. Retrieve a struct *TS3Variables* for this channel and fill it with the above mentioned functions ts3server_setVariableAsInt, ts3server_setVariableAsUInt64 and ts3server_setVariableAsString.

Retrieve a struct *TS3Variables* with

```
unsigned int ts3server_getChannelCreationParamsVariables(channelCreationParams, result);

struct TS3ChannelCreationParams* channelCreationParams;
struct TS3Variables** result;
```

- *channelCreationParams*

  Address of the struct *TS3ChannelCreationParams*, for which we want to retrieve the *TS3Variables*.

  result

  Address to be filled with the struct *TS3Variables* we want to retrieve.

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

Finally, after setting up server and channel parameters, create the virtual server including the channel tree in one step with

```
unsigned int ts3server_createVirtualServer2(virtualServerCreationParams, flags, result);

struct TS3VirtualServerCreationParams* virtualServerCreationParams;
```

```
enum VirtualServerCreateFlags flags;
uint64* result;
```

- *virtualServerCreationParams*

  Address of the struct *TS3VirtualServerCreationParams* with the creation parameters for this virtual server.

  flags

  Defines if the server password is passed as plaintext or already encrypted. If already encrypted, this is the password retrieved via the server variable *VIRTUALSERVER_PASSWORD*, which is returned as encrypted password. In this case you would specify the password on server creation as already encrypted to avoid the server encrypting it a second time.

  ```
  enum VirtualServerCreateFlags{
      VIRTUALSERVER_CREATE_FLAG_NONE                = 0x0000,
      VIRTUALSERVER_CREATE_FLAG_PASSWORDS_ENCRYPTED = 0x0001,
  };
  ```

  result

  Address to be filled with the created virtual server ID.

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;
```

- *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_VERSION_SIGN,     //internal use
  CLIENT_SECURITY_HASH,           //SDK only: Hash is provided by an outside source. A channel will
```

```
                                //use the security salt + other client data to calculate a hash,
        //which must be the same as the one provided here.
  CLIENT_ENCRYPTION_CIPHERS,      //SDK only: list of available ciphers send to the server
  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`.

- *CLIENT_SECURITY_HASH*

  Contains client security hash (optional feature). This hash is used to check if this client is allowed to enter specified channels with a matching *CHANNEL_SECURITY_SALT*. Motivation is to enforce clients joining a server with the specific identity, nickname and metadata.

  Please see Security salts and hashes for details.

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;
```

- *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;
```

- *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, result);

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

- *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_SECURITY_SALT,        //Sets the options+salt for security hash (SDK only)
  CHANNEL_DELETE_DELAY,         //How many seconds to wait before deleting this channel
  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: Codec used for this channel:

  - 0 - Speex Narrowband (8 kHz)
  - 1 - Speex Wideband (16 kHz)
  - 2 - Speex Ultra-Wideband (32 kHz)
  - 3 - Celt (Mono, 48kHz)
  - 4 - Opus Voice (Mono, 48khz)
  - 5 - Opus Music (Stereo, 48khz)

- *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).

- *CHANNEL_SECURITY_SALT*

  Contains the channels security salt (optional feature). When a client connects, the clients hash value in *CLIENT_SECURITY_HASH* is check against the channel salt to allow or deny the client to join this channel. Motivation is to enforce clients joining a server with the specific identity, nickname and metadata.

  Please see Security salts and hashes for details.

- *CHANNEL_DELETE_DELAY*

  This parameter defines how many seconds the server waits until a temporary channel is deleted when empty.

  When a temporary channel is created, a timer is started. If a user joins the channel before the countdown is finished, the channel is not deleted. After the last person has left the channel, the countdown starts again. *CHANNEL_DELETE_DELAY* defines the length of this countdown in seconds.

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;
```

- *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;
```

- *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_ENCRYPTION_CIPHERS,    //SDK only: list of ciphers that can be used for encryption
  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.

- *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.

- *VIRTUALSERVER_ENCRYPTION_CIPHERS*

  Comma-separated list of ciphers that are used for encrypting the connection. The server uses the left most cipher in *VIRTUALSERVER_ENCRYPTION_CIPHERS* that is also defined in *CLIENT_ENCRYPTION_CIPHERS* of the connecting client.

  Possible values are:

  ```
  "AES-128"
  "AES-256"
  ```

  Default is "AES-256,AES-128".

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);
```

In addition to the virtual server properties in the `public_definitions.h` header there are extended properties used for filetransfer found in a seperate header file `public_sdk_definitions.h`.

```
enum VirtualServerPropertiesSDK {
    VIRTUALSERVER_FILEBASE=24,                      //not available to clients, stores the folder used for file tra
    VIRTUALSERVER_MAX_DOWNLOAD_TOTAL_BANDWIDTH = 29, //only available on request (=> requestServerVariables)
    VIRTUALSERVER_MAX_UPLOAD_TOTAL_BANDWIDTH= 30,    //only available on request (=> requestServerVariables)
    VIRTUALSERVER_LOG_FILETRANSFER=64,
};
```

- *VIRTUALSERVER_FILEBASE*

  Base folder where to the file storage is located on this virtual server, see `ts3server_enableFileManager`. The filebase can be queried or set (before initializing filetransfer with `ts3server_enableFileManager`) with this property.

- *VIRTUALSERVER_MAX_DOWNLOAD_TOTAL_BANDWIDTH*

  Defines the maximum allowed bandwidth for downloadload. Set this property before creating the server with `ts3server_initServerLib`.

- *VIRTUALSERVER_MAX_UPLOAD_TOTAL_BANDWIDTH*

  Defines the maximum allowed bandwidth for upload. Set this property before creating the server with `ts3server_initServerLib`.

- *VIRTUALSERVER_LOG_FILETRANSFER*

  Set to true to enable logging filetransfer actions to the logfile.

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;
```

- *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*

In addition to the common connection properties, if filetransfer is enabled, the following filetransfer extra connection properties are available.

- *CONNECTION_FILETRANSFER_BANDWIDTH_SENT*

  How many bytes per second are currently being sent by file transfers

- *CONNECTION_FILETRANSFER_BANDWIDTH_RECEIVED*

  How many bytes per second are currently being received by file transfers

- *CONNECTION_FILETRANSFER_BYTES_RECEIVED_TOTAL*

  How many bytes we received in total through file transfers

- *CONNECTION_FILETRANSFER_BYTES_SENT_TOTAL*

  How many bytes we sent in total through file transfers

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;
```

- *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;
```

- *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.

Alternative way to create a new channel

There is an alternative API available for channel creation, which is very similar to the alternative virtual server creation API. The idea is to create a *`TS3ChannelCreationParams`*, query the attached struct *`TS3Variables`*, fill it with desired parameters and finally call `ts3server_createChannel`.

Note

For a complete example please see the “server_creation_params” sample code in the SDK package.

First, create a *`TS3ChannelCreationParams`* with

```
unsigned int ts3server_makeChannelCreationParams(result);

struct TS3ChannelCreationParams** result;
```

- *`result`*

  Address of a variable that receives the created struct *`TS3ChannelCreationParams`*.

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

Once we have a struct *`TS3ChannelCreationParams`* for this channel, we can start to fill it in two steps. Step 1 is setting the essential data, step 2 is setting optional additional data.

Essential parameters are channel parent ID and channel ID. Set them with

```
unsigned int ts3server_setChannelCreationParams(channelCreationParams, channelPar-
entID, channelID);

struct TS3ChannelCreationParams* channelCreationParams;
uint64 channelParentID;
uint64 channelID;
```

- *`channelCreationParams`*

  Address of the struct *`TS3ChannelCreationParams`*, where the parameters should be set.

channelParentID

ID of the parent channel. Set 0 to create this as top-level channel.

channelID

ID of the channel to create. This allows you to setup a complete channel tree with predefined channel IDs, unlike the legacy way to create channels where the channel ID is automatically assigned by the server.

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

After setting the essential parameters, we can set optional additional channel parameters. Retrieve a struct *TS3Variables* for this channel and fill it with ts3server_setVariableAsInt, ts3server_setVariableAsUInt64 and ts3server_setVariableAsString, as described here.

Retrieve a struct *TS3Variables* with

```
unsigned int ts3server_getChannelCreationParamsVariables(channelCreationParams, result);

struct TS3ChannelCreationParams* channelCreationParams;
struct TS3Variables** result;
```

- *channelCreationParams*

  Address of the struct *TS3ChannelCreationParams*, for which we want to retrieve the *TS3Variables*.

  result

  Address to be filled with the struct *TS3Variables* we want to retrieve.

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

Finally, after setting up channel parameters, create the channel in one step with

```
unsigned int ts3server_createChannel(serverID, channelCreationParams, flags, result);

uint64 serverID;
struct TS3ChannelCreationParams* channelCreationParams;
enum ChannelCreateFlags flags;
uint64* result;
```

- *channelCreationParams*

  Address of the struct *TS3ChannelCreationParams* with the creation parameters for this .

  flags

Defines if the channel password is passed as plaintext or already encrypted. If already encrypted, this is the password retrieved via the channel variable *CHANNEL_PASSWORD*, which is returned as encrypted password. In this case you would specify the password on channel creation as already encrypted to avoid it being encrypted automatically a second time.

```
enum ChannelCreateFlags{
    CHANNEL_CREATE_FLAG_NONE                = 0x000,
    CHANNEL_CREATE_FLAG_PASSWORDS_ENCRYPTED = 0x001,
};
```

result

Address to be filled with the created channel ID.

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

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;
```

- *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;
```

- *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;
```

- *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;
```

- *`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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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, textMessage);

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

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
```

- *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;
    }
}
```

Custom passwords

The TeamSpeak SDK has the optional ability to do custom password handling. This allows the possibility to check TeamSpeak server and channel passwords against an outside datasources, like LDAP or other databases.

To implement custom password, both server and client need to add custom callbacks, which will be spontaneously called whenever a password check is done in TeamSpeak. The SDK developer can implement own checks to validate the password instead of using the TeamSpeak built-in mechanism.

Both Server and Client Lib can implement the following callback to encrypt a user password. This function is called in the Server Lib when a virtual server or channel password are set.

This can be used to hash the password in the same way it is hashed in the outside data store. Or just copy the password to send the clear text to the server.

```
void onClientPasswordEncrypt(serverID, plaintext, encryptedText, encryptedTextByte-
Size);

uint64 serverID;
const char* plaintext;
char* encryptedText;
int encryptedTextByteSize;
```

- *serverID*

  ID of the server the password call occured

- *plaintext*

  The plaintext password

- *encryptedText*

  Fill with your custom encrypted password. Must be a 0-terminated string with a size not larger than *encrypted-TextByteSize*.

- *encryptedTextByteSize*

  Size of the buffer pointed to by *encryptedText*.

Implement this callback in the server to check the password provided when a client connects to this server against an outside database. The callback is called whenever a password check if performed, even if no password is set.

```
unsigned int onCustomServerPasswordCheck(serverID, client, password);

uint64 serverID;
const struct ClientMiniExport* client;
const char* password;
```

- *serverID*

  ID of the server on which a client password is checked when the client connects to this server

- *client*

  Identifies the connecting client

- *password*

  Password the client is using. This parameter is an empty string ("") if no password is set.

The function should return *ERROR_ok* if the password is verified, *ERROR_server_invalid_password* if not verified or *ERROR_invalid_param* if the password is in an invalid form.

Implement this callback in the server to check the password provided when a client enters a password-protected channel against an outside database. The callback is called whenever a password check if performed, even if no password is set.

```
unsigned int onCustomChannelPasswordCheck(serverID, client, password);

uint64 serverID;
const struct ClientMiniExport* client;
const char* password;
```

- *serverID*

  ID of the server on which a client password is checked when the client enters a password-protected channel

- *client*

  Identifies the client

- *password*

  Password the client is using. This parameter is an empty string ("") if no password is set.

The function should return *ERROR_ok* if the password is verified, *ERROR_server_invalid_password* if not verified or *ERROR_invalid_param* if the password is in an invalid form.

Custom permissions

The TeamSpeak SDK offers an optional custom permission system, with which SDK users can gain more control over allowed user actions on a TeamSpeak server. The custom permissions system is implemented in the Server Lib by adding callback functions which will be called, if implemented, when a certain user action occurs. In this callback the developer can allow or deny the action. If a callback is not implemented, the action will be allowed by default.

The callbacks should return *ERROR_ok* to allowe the action or *ERROR_permission* to deny it.

Example code to check if a client can connect to the server:

```
// Create the function pointer passed to ts3server_initServerLib
funcs.permClientCanConnect = onPermClientCanConnect;

// Custom callback
unsigned int onPermClientCanConnect(uint64 serverID, const struct ClientMiniExport* client) {
    // Forbid client with nickname "client" to connect
    if(strcmp(client->nickname, "client") == 0) {
        return ERROR_permissions;  // Deny
    }
    return ERROR_ok;  // Allow
}
```

Please see the `server_permissions` example for a demonstration of this mechanism.

Control if a client can connect to the server:

```
unsigned int permClientCanConnect(serverID, client);

uint64 serverID;
const struct ClientMiniExport* client;
```

Control if a client can access channel descriptions:

```
unsigned int permClientCanGetChannelDescription(serverID, client);

uint64 serverID;
const struct ClientMiniExport* client;
```

Control if a updating a client variable is allowed:

```
unsigned int permClientUpdate(serverID, clientID, variables);

uint64 serverID;
anyID clientID;
struct VariablesExport* variables;
```

Control if kicking a client from a channel is allowed:

```
unsigned int permClientKickFromChannel(serverID, client, toKickCount, toKickClients,
reasonText);

uint64 serverID;
const struct ClientMiniExport* client;
int toKickCount;
struct ClientMiniExport* toKickClients;
const char* reasonText;
```

Control if kicking a client from the server is allowed:

```
unsigned int permClientKickFromServer(serverID, client, toKickCount, toKickClients,
reasonText);

uint64 serverID;
const struct ClientMiniExport* client;
int toKickCount;
struct ClientMiniExport* toKickClients;
const char* reasonText;
```

Control if moving a client to a channel is allowed:

```
unsigned int permClientMove(serverID, client, toMoveCount, toMoveClients, newChannel,
reasonText);

uint64 serverID;
const struct ClientMiniExport* client;
int toMoveCount;
struct ClientMiniExport* toMoveClients;
uint64 newChannel;
const char* reasonText;
```

Control if moving a channel is allowed:

```
unsigned int permChannelMove(serverID, client, newChannel, newParentChannelID);

uint64 serverID;
const struct ClientMiniExport* client;
uint64 newChannel;
uint64 newParentChannelID;
```

Control if sending a text message is allowed:

```
unsigned int permSendTextMessage(serverID, client, targetMode, targetClientOrChan-
nel, textMessage);

uint64 serverID;
const struct ClientMiniExport* client;
anyID targetMode;
uint64 targetClientOrChannel;
const char* textMessage;
```

Control if requesting the server connection info is allowed:

```
unsigned int permServerRequestConnectionInfo(serverID, client);

uint64 serverID;
const struct ClientMiniExport* client;
```

Control if requesting the client connection info is allowed:

```
unsigned int permSendConnectionInfo(serverID, client, mayViewIpPort, targetClient);

uint64 serverID;
const struct ClientMiniExport* client;
int* mayViewIpPort;
```

```
const struct ClientMiniExport* targetClient;
```

Control if creating a channel with the given channel variables is allowed:

```
unsigned int permChannelCreate(serverID, client, parentChannelID, variables);

uint64 serverID;
const struct ClientMiniExport* client;
uint64 parentChannelID;
struct VariablesExport* variables;
```

Control if editing the given channel variables of a channel is allowed:

```
unsigned int permChannelEdit(serverID, client, channelID, variables);

uint64 serverID;
const struct ClientMiniExport* client;
uint64 channelID;
struct VariablesExport* variables;
```

Control if deleting a channel is allowed:

```
unsigned int permChannelDelete(serverID, client, channelID);

uint64 serverID;
const struct ClientMiniExport* client;
uint64 channelID;
```

Control if subscribing a channel is allowed:

```
unsigned int permChannelSubscribe(serverID, client, channelID);

uint64 serverID;
const struct ClientMiniExport* client;
uint64 channelID;
```

Security salts and hashes

As an optional security feature, the TeamSpeak SDK offers to restrict access of clients to specific channels by using a salt and hash mechanism. The motivation here is to enforce clients to use a specific identity, nickname and metadata when they connect to the TeamSpeak server.

In the server, a security salt is created over a clients unique data by calling `ts3server_createSecuritySalt`. This created salt is then attached to a channel during channel creation or by editing existing channels by setting the channel variable *CHANNEL_SECURITY_SALT*. When this channel variable is set, when a client enters the channel, the clients

CLIENT_SECURITY_HASH variable is checked against the clients data (unique id, optionally nickname and meta_data) using the salt. If the hash is not correct, the client is not allowed to enter the channel.

The clients hash value is calculated by the server calling `ts3server_calculateSecurityHash`. This security hash has to be transmitted to the client by ways outside of the TeamSpeak SDK. The client will set the hash in its *CLIENT_SECURITY_HASH* variable.

To create a security salt for the channel, call:

```
unsigned int ts3server_createSecuritySalt(options, salt, saltByteSize, securitySalt);

int options;
void* salt;
int saltByteSize;
char** securitySalt;
```

- *options*

  Combination of (OR'ed)

  ```
  SECURITY_SALT_CHECK_NICKNAME  -> means nickname will be used in security hash
  SECURITY_SALT_CHECK_META_DATA -> means metadata will be used in security hash
  ```

- *salt*

  Pointer to random data. This should be *good* random data, like cryptographic random.

- *saltByteSize*

  Size of the random data. Larger is better.

- *securitySalt*

  Pointer that receives the salt. Needs to be freed.

To create a security hash for a client, call:

```
unsigned int ts3server_calculateSecurityHash(securitySalt, clientUniqueIdentifier, clientNickName, clientMetaData, securityHash);

const char* securitySalt;
const char* clientUniqueIdentifier;
const char* clientNickName;
const char* clientMetaData;
char** securityHash;
```

- *securitySalt*

The channels salt data.

- *clientUniqueIdentifier*

  Unique identifier of the client we want to calculate the hash for.

- *clientNickName*

  Clients nickname

- *clientMetaData*

  Clients meta data

- *securityHash*

  Pointer that receives the hash. Needs to be freed.

Miscellaneous functions

Freeing memory

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

```
unsigned int ts3server_freeMemory(pointer);

void* pointer;
```

- *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.

Setting the log level

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

```
unsigned int ts3server_setLogVerbosity(logVerbosity);

enum LogLevel logVerbosity;
```

- *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.

Disabling protocol commands

SDK users can opt to disable specific protocol commands in a TeamSpeak server instance, so clients are unable to call these commands. The server can still issue disabled commands by calling the appropriate Server Lib functions.

This is an optional features. If certain commands are not disabled specifically, all commands are enabled by default.

For example, a SDK user may decide that clients should not be able to delete channels on a TeamSpeak server and implement this action in the Server Lib only. So he disabled the *CLIENT_COMMAND_requestChannelDelete* protocol command in the server and all client calls to ts3client_requestChannelDelete will be rejected by the server on the protocol level.

To disable specific protocol commands call:

```
unsigned int ts3server_disableClientCommand(clientCommand);

int clientCommand;
```

- *clientCommand*

  The command to disable in the server. Can be one of the following:

```
enum ClientCommand {
 CLIENT_COMMAND_requestConnectionInfo        =  0,
 CLIENT_COMMAND_requestClientMove            =  1, -> disables any channel switching requested by the client. (ser
 CLIENT_COMMAND_requestXXMuteClients         =  2, -> disable muting/unmuting clients
 CLIENT_COMMAND_requestClientKickFromXXX     =  3, -> disable kick requests from clients
 CLIENT_COMMAND_flushChannelCreation         =  4, -> disable creating new channels
 CLIENT_COMMAND_flushChannelUpdates          =  5, -> disables editing(changing) channel
 CLIENT_COMMAND_requestChannelMove           =  6, -> disbale moving channels around
 CLIENT_COMMAND_requestChannelDelete         =  7, -> disable deleting channels
 CLIENT_COMMAND_requestChannelDescription    =  8, -> disable getting channel description
```

```
   CLIENT_COMMAND_requestChannelXXSubscribeXXX =  9, -> disable subscriptions
   CLIENT_COMMAND_requestServerConnectionInfo  = 10,
   CLIENT_COMMAND_requestSendXXXTextMsg        = 11, -> disable sending text messages
      CLIENT_COMMAND_filetransfers               = 12, -> disable client filetransfer commands
      CLIENT_COMMAND_ENDMARKER
  };
```

If you want to disable multiple commands, call this function multiple times with one command per call.

There is no enable command. We recommend doing this call soon after calling `ts3server_initServerLib`.

Filetransfer

The TeamSpeak SDK includes the ability to support filetransfer, like the regular TeamSpeak server and client offer. The server can function as a file storage, which can be accessed by clients who can up- and download files. Files are stored on the filesystem where the server is running.

General filetransfer definitions and structures are described in the headers `public_sdk_definitions.h` and `server_commands.h`, which are required to include if you want to use the filetransfer feature.

In addition to the standard virtual server properties there are extented filetransfer properties available.

Filetransfer bandwidth statistics can be queried with special filetransfer connection properties.

The availability of filetransfer in the TeamSpeak server can be controlled by the following function, which should be called right after `ts3server_initServerLib` to initialize filetransfer.

```
unsigned int ts3server_enableFileManager(filebase, ips, port, downloadBandwitdh, uploadBandwidth);
```

```
const char* filebase;
const char** ips;
int port;
uint64 downloadBandwitdh;
uint64 uploadBandwidth;
```

- *filebase*

  The base folder where to save the files to. If this is initalized to "sdk_files", we would get the following directory structure:

  - All files for virtual server with id 1 will be in “sdk_files/virtualserver_1”.
  - Files in channel 1 in virtual server 1 will be in “sdk_files/virtualserver_1/channel_1”
  - These directories will automatically be created by the server if they do not exist. It is the responsibility of the application (not serverlib) to delete these directories when they are no longer needed.

- *ips*

  Array of IPs (IPv4 and IPv6 are supported) to bind to. This can be NULL, in which case we bind to “0.0.0.0, ::”. Otherwise pass a NULL-terminated array of strings (IPs, not domainnames).

- *port*

  Specifies the TCP port used for filetransfer. Free to choose between 1 and 65536. TeamSpeak defaults to 30033.

- *downloadBandwidth*

  Download limit in bytes/s or *BANDWIDTH_LIMIT_UNLIMITED*

- *uploadBandwidth*

  Upload limit in bytes/s or *BANDWIDTH_LIMIT_UNLIMITED*

Callbacks

The serverlib notifies about ongoing filetransfers with an event, which is called everytime a file upload or download has finished:

```
void onFileTransferEvent(data);

const struct FileTransferCallbackExport* data;
```

- *data*

  Data structure containing values describing the finished filetransfer event:

  - *clientID*

    ID of the client who triggerd the filetransfer

  - *transferID*

    The local filetransfer ID

  - *remoteTransferID*

    The remote filetransfer ID

  - *status*

    Info on the state of the transfer, defined by the struct *FileTransferState*:

    ```
    enum FileTransferState {
        FILETRANSFER_INITIALISING = 0,
        FILETRANSFER_ACTIVE,
        FILETRANSFER_FINISHED,
    };
    ```

  - *statusMessage*

    Info on the state in text form

  - *remotefileSize*

Size of the transfer file

- *bytes*

  Transferred bytes

- *isSender*

  True, if the server is sending the file, false if the server is receiving the file.

To change the filename or path of a file transfer, the following callback can be implemented. This is optional, when modifying path or name is not required, simply do not implement this callback. When this function is called, default path and name values are already filled in the passed structs.

```
unsigned int onTransformFilePath(serverID, invokerClientID, original, result);
```

```
uint64 serverID;
anyID invokerClientID;
const struct TransformFilePathExport* original;
struct TransformFilePathExportReturns* result;
```

- *serverID*

  ID of the virtual server on which the event is called

- *invokerClientID*

  ID of the client which invoked the file transfer

- *original*

  Struct TransformFilePathExport, defining the original file transfer path and name:

  ```
  struct TransformFilePathExport {
      uint64 channel;
      const char* filename;
      int action;
      int transformedFileNameMaxSize;
      int channelPathMaxSize;
  };
  ```

  - *channel*

    ChannelID if appropriate (not for *FT_INIT_SERVER*, where it defaults to 0)

  - *filename*

    Original file name

  - *action*

    Event was called for which action? (See below)

- *transformedFileNameMaxSize*

  Maximum size of transformedFileName (see result param)

- *channelPathMaxSize*

  Maximum size of channelPath (see result param)

- *result*

  Struct TransformFilePathExportReturns, target struct where the file transfer path or name can be changed:

```
struct TransformFilePathExportReturns {
    char* transformedFileName;
    char* channelPath;
    int logFileAction;
};
```

  - *transformedFileName*

    Pointer to string where the transformed filename is located. To change it, overwrite the content of where the pointer points, up to *original->transformedFileNameMaxSize* bytes.

  - *channelPath*

    Pointer to string where the channel path is located. To change it, overwrite the content of where the pointer points, up to *original->channelPathMaxSize* bytes.

  - *logFileAction*

    Set to 0 to not log to logfile, or 1 to log to file. *VIRTUALSERVER_LOG_FILETRANSFER* must also be true for logging to happen (see struct VirtualServerPropertiesSDK in `public_sdk_definitions.h`).

Return with *ERROR_ok* if there are no errors, or return an other error value.

When implementing this callback, depending on the state in *original->action* different fields in the *result* struct can be modified.

Possible states for *original->action* are:

- *FT_INIT_SERVER*

  When the virtual server is being is being created. The *result->channelPath* can be changed to create a different folder for the virtual server.

- *FT_INIT_CHANNEL*

  When a channel is being is being created. The *result->channelPath* can be changed to create a different folder for the channel.

- *FT_UPLOAD*

  When a file is being uploaded. Here *result->channelPath* and/or *result->transformedFileName* can be changed.

- *FT_DOWNLOAD*

When a file is being downloaded. Here *result->channelPath* and/or *result->transformedFileName* can be changed.

- *FT_DELETE*

  When a file is being deleted. Here *result->channelPath* and/or *result->transformedFileName* can be changed.

- *FT_CREATEDIR*

  When a folder is being created. Here *result->channelPath* and/or *result->transformedFileName* can be changed

- *FT_RENAME*

  When a file/folder is being renamed. This callback will be called 2 times, first for old, then for new name. Here *result->channelPath* and/or *result->transformedFileName* can be changed.

- *FT_FILELIST*

  Called when a listing of a folder is requested. Here *result->channelPath* and/or *result->transformedFileName* can be changed.

- *FT_FILEINFO*

  Called when file info is requested. Here *result->channelPath* and/or *result->transformedFileName* can be changed.

Permissions

The follow callbacks can be optionally implemented to control if various filetransfer actions are allowed or denied. If a callback is not implemented, actions are allowed by default.

If you want to use these functions, include `server_commands.h`.

Control if a client is allowed to upload a file:

```
unsigned int permFileTransferInitUpload(serverID, client, params);

uint64 serverID;
const struct ClientMiniExport* client;
const struct ts3sc_ftinitupload* params;
```

- *serverID*

  ID of the server where the file transfer is initiated.

- *client*

Summary client info:

```
struct ClientMiniExport {
    anyID ID;
    uint64 channel;
    const char* ident;
    const char* nickname;
};
```

- *params*

Summary file info (fileName/fileSize/channelID):

```
struct ts3sc_ftinitupload {
    struct ts3sc_meta_ftinitupload   m; /* message meta data */
    struct ts3sc_data_ftinitupload   d; /* message data */
};

struct ts3sc_data_ftinitupload {
    const char*  fileName;  /* The file name */
    uint64       fileSize;  /* The file size */
    uint64       channelID; /* The channel ID where the file is to be uploaded */
    int          overwrite; /* Set to 1 to overwrite files, 0 to prevent overwrites */
    int          resume;    /* Set to 1 to resume an existing upload, 0 to start new */
};
```

Return *ERROR_OK* if allowed or *ERROR_permissions_client_insufficient*/*ERROR_permissions*.

Control if a client is allowed to download a file:

```
unsigned int permFileTransferInitDownload(serverID, client, params);

uint64 serverID;
const struct ClientMiniExport* client;
const struct ts3sc_ftinitdownload* params;
```

- *serverID*

ID of the server where the file transfer is initiated.

- *client*

Summary client info

- *params*

Summary file info (fileName/channelID):

```
struct ts3sc_ftinitdownload {
    struct ts3sc_meta_ftinitdownload   m; /* message meta data */
    struct ts3sc_data_ftinitdownload   d; /* message data */
};

struct ts3sc_data_ftinitdownload {
     const char*  fileName;  /* The file name */
     uint64       channelID; /* The channel ID where the file is to be downloaded from */
```

```
};
```

Return *ERROR_OK* if allowed or *ERROR_permissions_client_insufficient*/*ERROR_permissions*.

Control if a client is allowed to request the file information:

```
unsigned int permFileTransferGetFileInfo(serverID, client, params);

uint64 serverID;
const struct ClientMiniExport* client;
const struct ts3sc_ftgetfileinfo* params;
```

- *serverID*

  ID of the server where the file info request is initiated.

- *client*

  Summary client info

- *params*

  Summary file info (fileName/channelID):

```
struct ts3sc_ftgetfileinfo {
    struct ts3sc_meta_ftgetfileinfo   m;      /* message meta data */
    struct ts3sc_data_ftgetfileinfo   d;      /* message data */
    int                               r_size; /* items in r */
    struct ts3sc_array_ftgetfileinfo* r;      /* message repeat data */
};

struct ts3sc_data_ftgetfilelist {
     uint64       channelID; /* The channel ID */
     const char*  path;      /* The path where to get the list for */
};

struct ts3sc_array_ftgetfileinfo {
     uint64 channelID;     /* The channel ID where the file is located */
     const char* fileName; /* The file name */
};
```

Return *ERROR_OK* if allowed or *ERROR_permissions_client_insufficient*/*ERROR_permissions*.

Control if a client is allowed to request a directory listing:

```
unsigned int permFileTransferGetFileList(serverID, client, params);

uint64 serverID;
const struct ClientMiniExport* client;
const struct ts3sc_ftgetfilelist* params;
```

- *serverID*

  ID of the server where the file list request is initiated.

- *client*

  Summary client info

- *params*

  Summary file info (path/channelID):

```
struct ts3sc_ftgetfilelist {
    struct ts3sc_meta_ftgetfilelist   m; /* message meta data */
    struct ts3sc_data_ftgetfilelist   d; /* message data */
};

struct ts3sc_data_ftgetfilelist {
     uint64       channelID; /* The channel ID */
     const char*  path;      /* The path where to get the list for */
};
```

Return *ERROR_OK* if allowed or *ERROR_permissions_client_insufficient*/*ERROR_permissions*.

Control if a client is allowed to delete one or more files:

```
unsigned int permFileTransferDeleteFile(serverID, client, params);

uint64 serverID;
const struct ClientMiniExport* client;
const struct ts3sc_ftdeletefile* params;
```

- *serverID*

  ID of the server where the file deletion is initiated.

- *client*

  Summary client info

- *params*

  Summary file info (channelID/array of filenames):

```
struct ts3sc_ftdeletefile {
    struct ts3sc_meta_ftdeletefile   m;      /* message meta data */
    struct ts3sc_data_ftdeletefile   d;      /* message data */
    int                              r_size; /* items in r */
    struct ts3sc_array_ftdeletefile* r;      /* message repeat data */
};

struct ts3sc_data_ftdeletefile {
     uint64  channelID; /* The channel ID where the file is to be deleted */
};

struct ts3sc_array_ftdeletefile {
```

```
      const char* fileName;  /* The file name to be deleted */
  };
```

Return *ERROR_OK* if allowed or *ERROR_permissions_client_insufficient*/*ERROR_permissions*.

Control if a client is allowed to create a directory:

```
unsigned int permFileTransferCreateDirectory(serverID, client, params);

uint64 serverID;
const struct ClientMiniExport* client;
const struct ts3sc_ftcreatedir* params;
```

- *serverID*

  ID of the server where the directory creation is initiated.

- *client*

  Summary client info

- *params*

  Summary file info (channelID/dirname):

```
struct ts3sc_ftcreatedir {
    struct ts3sc_meta_ftcreatedir   m; /* message meta data */
    struct ts3sc_data_ftcreatedir   d; /* message data */
};

struct ts3sc_data_ftcreatedir {
     uint64       channelID; /* The channel ID where the file is to be uploaded */
     const char*  dirname;   /* The directory name */
};
```

Return *ERROR_OK* if allowed or *ERROR_permissions_client_insufficient*/*ERROR_permissions*.

Control if a client is allowed to rename a file:

```
unsigned int permFileTransferRenameFile(serverID, client, params);

uint64 serverID;
const struct ClientMiniExport* client;
const struct ts3sc_ftrenamefile* params;
```

- *serverID*

  ID of the server where the file rename is initiated.

- *client*

Summary client info

- *params*

  Summary file info (fromChannelID/toChannelID/oldFileName/newFileName). Use *params->m.has_toChannelID* to check if *toChannelID* is valid (else it is a rename in the same channel)

```
struct ts3sc_ftrenamefile {
    struct ts3sc_meta_ftrenamefile   m; /* message meta data */
    struct ts3sc_data_ftrenamefile   d; /* message data */
};

struct ts3sc_data_ftrenamefile {
     uint64       fromChannelID; /* The channel ID where the file is located now */
     uint64       toChannelID;   /* The channel ID where the file is to be moved to */
     const char*  oldFileName;   /* The current file name */
     const char*  newFileName;   /* The new file name */
};
```

Return *ERROR_OK* if allowed or *ERROR_permissions_client_insufficient*/*ERROR_permissions*.

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 | | virtual server id 1 is running elsewhere, shutting down" 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@teamspeakusa.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 shut-down" 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

B

C

E

F

L

M

S

W