Client program interface

Table of Contents

Quick start

Installation and preparation for use

The P2 CGate library consists of the following components:

- Plaza-2 system libraries
- P2MQRouter message router
- cgate gate library
- cgate.h — a header file with API description

All these components are required for development using the P2 CGate library.

To begin development, it is necessary to install the components by the installer corresponding to your operation system. Depending on the operation system, the libraries and the header file will be installed either into default locations or into locations specified during installation process. In further instructions the installation folder will be specified as ‘CGATE_HOME’.

Important

Login to the Plaza-2 system and the application key are required for work with the library. Logins to the test system Plaza-2 and test keys are used for development — they may be used freely by any developer. Production logins and keys are used for production environment. Production keys may be obtained upon passing the certification procedure.

Building and running examples may be performed to verify whether installation has finished correctly and the system is ready for development. To do this, it is necessary to perform the following steps:

1. Configuration of the Plaza-2 router according to the available login (this activity is performed automatically if an interactive installer was used)

   Open router setting file P2MQRouter, which is usually called client_router.ini and type in login and password in the section [AS:NS]:

   ```
   [AS:NS]
   USERNAME=<your login>
   PASSWORD=<your password>
   ```

2. Building examples

   Examples are located in the CGATE_HOME\sdk\samples folder for Windows platform and in the /usr/share/doc/cgate-examples folder for Linux. Examples can be built with build scripts which vary depending on the used platform and programming language. For Linux OS, it is recommended to prepare a copy of examples in the home folder and to build them from this folder.

3. Running examples

   To run examples it is necessary to make sure that the P2MQRouter router is active and connected to the Plaza-2 network (by analysis of router messages). Also please make sure that the Plaza-2 libraries are accessible (it may be required to add the CGATE_HOME \p2bin directory to the PATH environment variable or to specify the CGATE_HOME\p2bin directory in your development environment), and also that configuration files are available. Examples are started without parameters.

Main objects

The library introduces a set of objects which are used to get access to different functions of the system. The main objects are:

Environment	Describes working environment of the library. This object exists in the single copy. It is intended for initialization and deinitialization of sub-systems, maintenance of operating logs and memory control.
Connection	Provides access to the connection with Plaza-2 router

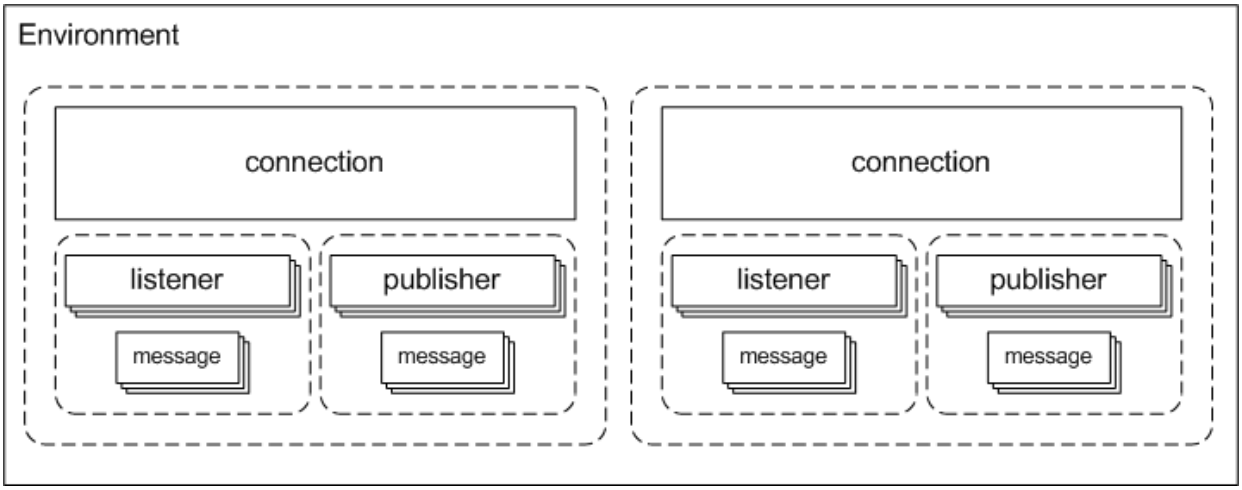
Message — Describes a message. Messages are used for representing any information which is sent and received by the user — data updating notifications, orders sent to the trading system, reports on orders execution, notifications on opening and closure of data streams.

Listener — Provides access to receiving of messages. This interface is used for receiving of all messages — updates of data streams, reports on orders execution — if you receive any message, you do it by means of the Listener object.

Publisher — Provides access to messages sending. Everything which is sent by your code is sent by means of one of the Publisher objects.

The Listener and Publisher objects are tied to particular connections. You may use numerous connections, numerous listeners and publishers depending on the architecture of your application; usually, connections for receiving updates of market information are separated from connections for sending orders.

The general scheme of the library objects within the client program is the following:



General environment may include several connections; each connection contains arbitrary number of listeners and publishers and each of them has a certain number of messages. In actual practice, the purpose of each connection and listeners and publishers linked with this connection usually depends on actual demands of the application.

Start-up and shutdown of the environment

To start work with the library, it is required to perform initialization of environment. Initialization is performed via the ‘env_open’ function:

```
CG_RESULT cg_env_open(const char* settings);
```

The function accepts string that describes parameters of the system. The string is a set of ‘KEY=VALUE’ pairs separated by a semicolon. Two parameters are available:

ini — Initialization file path. This file describes configuration of the library — journaling mode, etc.

Parameter setting may look like: "ini=conf/settings.ini", in this case the library will load configuration from the file conf/settings.ini

key — Client program identifier, which must be specified for work with the library. The key is used to get access to the Plaza-2 system — for the test system there is a set of predefined keys; for production — the key is generated upon passing the program certification procedure.

Failure of initialization may indicate a configuration error: the configuration file may be missed or installation integrity may be corrupted, etc. In case of this failure, there is no use trying to perform re-initialization of libraries; instead of this, you should shutdown your program and check the configuration.

If initialization fails or was not performed, it is not possible to work with the other functions of the library.

Initialization code of the system may look like:

```
result = cg_env_open("ini=ini/settings.ini;key=72395823576");
if (result != CG_ERR_OK)
{
    // display an error message and exit the program
    ...
    return;
}
```

Which means successful initialization of the library with the ‘configuration ini/settings.ini' file and the application key '72395823576'. The 'ini/settings.ini' file should be accessible via the specified path relatively to the current work directory at the moment of program start.

Deinitialization is performed before exiting the program by calling the 'env_close' function:

```
CG_RESULT cg_env_close(void);
```

The function performs deinitialization of sub-systems and closure of the operating log. This function must be always called in the end of program operation.

Working with connection

The ‘Connection’ object provides interaction with the Plaza-2 router for sending and receiving messages. These objects may be created in any quantity at any time during program operation with initialized environment; nevertheless, it is recommended to make connections at the start of program and to destroy them just before exiting.

Connection is created by calling ‘cg_conn_new’, for instance, in the following way:

```
cg_conn_t* conn;
result = cg_conn_new("p2tcp://127.0.0.1:4001;app_name=test", &conn);
```

In this example, a connection via the TCP/IP protocol with the Plaza-2 router on the port 4001 initiated on the same computer and with the application name ‘test’ created. Calling this function initializes connection object but doesn’t lead to actual establishment of connection.

Connection is established by calling the 'conn_open' function:

```
result = cg_conn_open(conn, 0);
```

, where ‘conn’ — the object initialized by the ‘cg_conn_new’ function call, and 0 (as the second parameter) means absence of connection opening call parameters.

Connection is closed by calling ‘conn_close’:

```
result = cg_conn_close(conn);
```

In this case, interaction with the Plaza-2 router is closed but the object remains initialized and may be reopened.

Object is destroyed by the ‘conn_destroy’ function:

```
result = cg_conn_destroy(conn);
```

Initialization of connection may fail when installation integrity is corrupted or when there is incorrect configuration, e.g. incorrect parameters have been specified. In this case, the best things to do are to shutdown program and analyze the configuration.

Opening of connection may fail with an error due to different reasons, e.g. the Plaza-2 router is not ready to service incoming connections, there is a failure in the communication channel, etc. Opening of connection should be performed in a cyclical manner since the next attempt of opening may become successful.

Example of the described behavior:

```
cg_conn_t* conn;
result = cg_conn_new("p2tcp://127.0.0.1:4001;app_name=test", &conn);
if (result != CG_ERR_OK)
{
    // failure of connection initialization
    // further work is impossible
    // report on the error and exit the program
    return;
}

// initialized object conn exists in this place,
// this object may be worked with – get its status, open, close

while (haveToExit()) // main loop of the program
{
    uint32_t state;
    result = cg_conn_getstate(conn, state); // get a status of connection
    if (result != CG_ERR_OK) // error in getting the connection status
    {
        // report the error and exit the program
        return;
    }
    switch (state)
    {
    case CG_STATE_CLOSED: // connection is closed, try to open
        result = cg_conn_open(conn, 0);
        // make a report in case of error
        break;
    case CG_STATE_ERROR: // connection is in the error state, it should be closed
        result = cg_conn_close(conn);
        // make a report in case of error
        break;
    case CG_STATE_ACTIVE: // connection is active, it may be worked with
```

```
        ...
    }
    ...
}
```

This cycle implements correct work with connection: if connection is closed, an attempt will be made to open it; if connection went to the error state, then it will be closed. Work with connection is performed when it is active.

This example uses the cg_conn_getstate function:

```
uint32_t state;
result = cg_conn_getstate(conn, state);
```

This function returns the state of the initialized ‘Connection’object. Messages may be sent and received only when the corresponding connection is in the ‘active’ state ('CG_STATE_ACTIVE').

Being in the active state, the connection requires periodical calling of the event processing the ‘conn_process’ function, which performs calling of user-defined callback functions and internal processing:

```
case CG_ACTIVE:
{
    result = cg_conn_process(conn, 0);
    if (result != CG_ERR_OK && result != CG_ERR_TIMEOUT)
    {
        // connection work is broken
        result = cg_conn_close(conn);
    }
    ...
    break;
}
```

The ‘conn_process’ function uses the second parameter as a time interval in milliseconds which is the time to wait for new event to take place within the connection framework. Awaiting the new calls, the ‘conn_process’ call is blocked. If there were no messages during the specified time interval, the function will return the ‘CG_ERR_TIMEOUT’ value — in this case, this value is not an error indicator and may be used, for instance, to indicate that there are no incoming messages and that program logics may pass to the next task.

Receiving data streams

Data streams are received by means of the ‘Listener’ objects. The ‘Listener’ object is created linked to connection with ‘cg_lsn_new’ call:

```
result = cg_lsn_new(conn, "p2repl://FORTS_FUTTRADE_REPL", dataCB, user_data, &lsn);
```

In this example, ‘lsn’ is initialized by the ‘Listener’ object, which is set for receiving of the ‘FORTS_FUTINFO_REPL’ data stream via the connection conn. Messages on data updates and on other events of the stream life cycle will be passed to the user-defined ‘dataCB’ callback function. When creating a subscription, it is possible to set different parameters including the client replication scheme; in this case, initialization of the object will be performed in the following way:

```
result = cg_lsn_new(conn,
               "p2repl://FORTS_FUTTRADE_REPL;scheme=|FILE|ini/futtrade.ini|FutTrade",
               dataCB, user_data, &lsn);
```

, where the scheme description file path and the section name of the corresponding ini-file are set in the ‘scheme’ parameter by the string of a special format.

Upon successful calling of the ‘cg_lsn_new’ function, the object goes into the initialized but non-active state. In fact, the stream is opened by calling the ‘cg_lsn_open’ function:

```
result = cg_lsn_open(lsn, 0);
```

In this example, the data stream is opened without parameters, which means that it will be opened with default parameters:

- number of the data scheme life is not set (equal to 0)
- revisions of all tables are equal to 0 which means that they will be received anew
- replication mode is selected as snapshot+online which leads to receiving of tables snapshot (or their full history) and then switching to online data receiving

Parameters are specified as a string:

```
result = cg_lsn_open(lsn, "mode=online");
```

In this case, the stream will be opened on the online mode which skips the initial snapshot stage. See description of the ‘cg_lsn_open’ function for detailed information on supported parameters.

The ‘cg_lsn_open’ function may return the error code in different cases: temporary unavailability of the stream, malfunction of the channel operation. For correct operation it is required to perform cyclical opening of flows.

The stream is closed by calling the ‘cg_lsn_close’ function:

```
result = cg_lsn_close(lsn);
```

In this case, the listener is disconnected from data receiving, and updates of this stream are no longer transmitted through the connection; the object itself remains initialized and may be reopened, including opening with different parameters.

The object is destroyed by calling ‘cg_lsn_destroy’:

```
result = cg_lsn_destroy(lsn);
```

After that, the ‘lsn’ object is released, and the further work with this object becomes impossible.

For correct receiving the data updates, the ‘Listener’ object must call the ‘conn_process’ function for the connection to the object it is tied to. Data receiving frequency does not exceed the frequency of the ‘conn_process’ function calling. Therefore, in order to provide maximum data receiving speed it is required to provide maximum possible frequency of the ‘conn_process’ function calling for desired connections.

Receiving of data and occurrence of other events in the life cycle of the data stream is accompanied by calling of the user-defined ‘lsn_new’ callback function which looks like:

```
typedef CG_RESULT (*CG_LISTENER_CB)(cg_conn_t* conn,
                                    cg_listener_t* listener,
                                    struct cg_msg_t* msg,
                                    void* data);
```

The following information is transferred to the callback function:

- ‘conn’ — connection the listener is tied to
- ‘listener’ — the 'Listener' object
- ‘msg’ — received message
- ‘data’ — user data which were transferred as of the moment of the ‘lsn_new’ function calling

The ‘msg’ message which is transferred to the user-defined function is generally described by the following structure:

```
struct cg_msg_t
{
    uint32_t type;     // Message type
    size_t data_size;  // Amount of data
    void* data;        // Pointer to data
};
```

Any message, which is delivered to the user-defined function, has the listed fields in any case.

Particular message type is identified with the ‘type’ field analysis. The following message types are used when data stream is received:

CG_MSG_OPEN	The message is delivered at the moment of data stream activation. This event surely occurs before receiving of any data on this subscription. For data streams, delivery of the message means that the data scheme was agreed and is ready to use (for more details see Data schemes). This message does not contain additional data, and its ‘data’ and ‘data_size’ fields are not used.
CG_MSG_CLOSE	The message is delivered at the moment of data stream closure. Delivery of the message means that the stream was closed by the user or the system. This message does not contain additional data, and its ‘data’ and ‘data_size’ fields are not used.
CG_MSG_TN_BEGIN	Means the moment when receiving of the next data block starts. Along with the next message, may be used by the program logic for data integrity control. This message does not contain additional data, and its ‘data’ and ‘data_size’ fields are not used.
CG_MSG_TN_COMMIT	Means the moment when receiving of the next data block is completed. By the moment this message is delivered, it may be safely assumed that data received under this subscription are consistent and reflect the inter-synchronized tables. This message doesn’t contain additional data, and its ‘data’ and ‘data_size’ fields are not used.
CG_MSG_STREAM_DATA	The message indicating delivery of stream data. The ‘data_size’ field contains the amount of data received; ‘data’ indicates the data themselves. The message itself contains additional fields which are described by the ‘rtscg_msg_streamdata_t’ structure. See the information presented below in this section for more details on data receiving.
CG_MSG_P2REPL_ONLINE	Stream switching to the online mode — it means that receiving of the initial snapshot was completed, and the ‘CG_MSG_STREAM_DATA’ messages below will bear online data. This message does not contain additional data, and its ‘data’ and ‘data_size’ fields are not used.
CG_MSG_P2REPL_LIFENUM	The scheme life number was changed. This message means that previous data, which were received regarding the stream are not up-to-date and should be deleted. This will be accompanied by retranslation of data on the new data scheme life number. The ‘data’ field of the message indicates an integer value containing the new scheme life number; the ‘data_size’ field indicates the size of the integral type.

CG_MSG_P2REPL_CLEARDELETED — Mass deletion of outdated data was performed. The ‘data’ field of the message indicates the ‘cg_data_cleardeleted_t’ structure, which indicates the number of table and the number of revision — data in this table issued prior to this revision are deemed to be deleted.

CG_MSG_P2REPL_REPLSTATE — The message indicates the state of data stream; it is sent before closure of the stream. The ‘data’ field of the message indicates the line, which indicates the encoded state of the data stream as of the moment the message is delivered — the data scheme, table revision numbers and the scheme life number are saved. This line may be transferred for calling of the cg_lsn_open function as the ‘replstate’ parameter on the same stream on the next time which will provide continuation of data receiving upon shutdown of the stream.

When the ‘RTSCG_MSG_P2REPL_DATA’ event occurs, the ‘msg’ parameter of the user-defined callback function contains pointer to the extended data structure:

```
struct cg_msg_streamdata_t
{
    uint32_t type;   // Message type. Always RTSCG_MSG_P2REPL_DATA for this message
    size_t data_size;  // Data amount
    void* data;        // Data indicator
    size_t msg_index;  // Number of the message description in the active scheme
    uint32_t msg_id;   // Unique identifier of the message type
    const char* msg_name; // Message name in the active scheme
    int64_t rev;          // Record revision
    const uint8_t* nulls; // Byte array containing 1 for each field which is considered to be NULL.
};
```

The extended structure is accessed in the following way:

```
RTSCG_RESULT dataCallback(rtscg_conn_t* conn,
                          rtscg_listener_t* listener,
                          struct rtscg_msg_t* msg,
                          void* data)
{
    switch(msg->type)
    {
        case CG_MSG_STREAM_DATA:
        {
             // bringing the indicator to the extended structure
            cg_msg_streamdata_t* replmsg = (cg_msg_streamdata_t*)msg;
            // extended structure may be used here
            ...
        }
        ...
    }
}
```

This structure may be used to determine the table number and its name in the data scheme — this information is accessible in the ‘msg_index’ and ‘msg_name’ fields of the structure. For the Plaza-2 data stream, the ‘msg_id’ field is not used and its value is 0. The ‘rev’ field contains record revision (update number) in the table, and the ‘nulls’ field may contain the indicator of byte array which determines whether the record contains a particular field or not.

Data, which the ‘data’ message indicator refers to, are structured according to the data scheme applied in this subscription. See the next section for details on data schemes and access to desired record fields.

Working with data schemes

Any data received or sent in the course of client program interaction with the trading system have a particular structure. Data schemes are used to describe the structure of particular messages.

Data scheme describes a set of possible messages for the selected data channel (subscription or publishing), fields and types of these messages and also define the rules of access to these data. A data scheme is described by the following structure:

```
struct cg_scheme_desc_t {
  // Scheme type
  uint32_t scheme_type;

  // Scheme features
  uint32_t features;

  // Number of messages in the scheme
  size_t num_messages;

  // Indicator of the messages description list
  struct cg_message_desc_t* messages;
};
```

The only one scheme type is currently available; this type corresponds to the identifier ‘1’ — data are stored in the binary form with the 4-byte alignment without support of optional fields.

The ‘features’ field describes available information in the scheme — this field may be used to determine whether default values were set for fields in this scheme, whether fields or messages have descriptions, etc. This is accomplished by the ‘CG_SCHEME_BIN_*’ constants.

The ‘num_messages’ field defines the number of messages in the scheme, and the ‘messages’ field indicates the first message. Messages represent the main object describing particular data structures and are used in all types of subscriptions and publishing; for instance, for the Plaza-2 replication, messages describe events of data updates in tables.

Each message is described by the following structure:

```
struct cg_message_desc_t {
    // indicator of the next message
    struct cg_message_desc_t* next;

    // message block size
    size_t size;

    // Number of fields in the message
    size_t num_fields;

    // Indicator of field descriptions array
    struct cg_field_desc_t* fields;

    // Message identifier
    // May be equal to 0, if the message has no identifier
    uint32_t id;

    // Message name indicator
    // May be NULL – in this case the message has no name
    char *name;

    // Message description indicator
    // May be NULL - in this case, the message has no description
    char *desc;

    // line with hints
    char* hints;

    // number of message indices
    size_t num_indices;

    // First index indicator
    struct cg_index_desc_t* indices;
};
```

The ‘next’ field indicates the next message in the scheme or contains the 'NULL' value, that indicates the last message. Therefore, messages are arranged in the linked list and may be accessed by the following cycle:

```
cg_scheme_desc_t* schemedesc; // initialized indicator of data schemes
for (cg_message_desc_t* msgdesc = schemedesc->messages;
     msgdesc; msgdesc = msgdesc->next)
{
    // here it is possible to work with message description
    // which is contained in msgdesc

    ...
}
```

The ‘size’ field of the message description structure specifies the block size in bytes, required for storing the entire message data. The ‘num_fields’ field indicates the number of fields in the message, and ‘fields’ indicates the first message field.

The ‘id, ‘name’ and ‘desc’ fields contain the message identifier along with its name and its description. The message may have no identifier, name or description if a particular scheme does not describe these values.

The ‘hints’ field contains parameters which may be used for automatic setting of the program for a particular type or method of data updating. At the present moment, the data scheme describing commands of the FORTS trading system contains the ‘request’ and ‘reply’ hints, advising the messages to be sent and the messages to be received. The ‘hints’ field may contain several hints separated by ‘;’ symbol.

The ‘num_indices’ field contains the number of indices, and the ‘indices’ field indicates the first index. The first index in the list is always the unique primary key.

Indices are described by the following structure:

```
struct cg_index_desc_t {
    // indicator of the next index
    struct cg_index_desc_t * next;

    // number of fields in the key
    size_t num_fields;

    // indicator of the first field description in the key
    struct cg_indexfield_desc_t* fields;

    // key name
    char* name;

    // key description
    char* desc;

    // line with hints
    char* hints;
};
```

The ‘next’ field indicates the next index in the scheme or contains the NULL value, indicating the last index.

The ‘num_fields’ field indicates the number of fields in the index.

The ‘fields’ field indicates the first field in the index.

The ‘name’ and ‘desc’ fields contain the index name and its description.

The ‘hints’ field contain hints for the index. For example, ‘unique’.

The index fields are described by the following structure:

```
struct cg_indexfield_desc_t {
    // indicator of the next key field description
    struct cg_indexfield_desc_t* next;

    // field indicator
    struct cg_field_desc_t* field;

    // sorting order
    uint32_t sort_order;
};
```

The ‘next’ field indicates the next field in the index or contains the ‘NULL’ value, indicating the last field.

The ‘field’ field indicates the structure describing the scheme fields.

The ‘sort_order’ field specifies the sorting order: 0 - ascending, 1 - descending.

The message fields are described by the following structure:

```
// Message field description
struct cg_field_desc_t {
    // pointer to the next field
    struct cg_field_desc_t* next;

    // Field identifier
 // Can be 0 in case there is no field id
    uint32_t id;

    // Field name
 // Can be NULL - in this case field has no name
    char* name;

    // Field description
 // Can be NULL - in this case field has no description
    char* desc;

    // Field type
    char*  type;

    // Value size of this field
    size_t size;

    // Offset from the message beginning
```

```
    size_t offset;

    // Indicator of the default field value.
    // Indicates the buffer of the ‘size’ size which stores data in the ‘type’ format
    // If null, then there is no default value

    void* def_value;

    // Indicator of the list of field values
    struct cg_field_value_desc_t* values;
};
```

The ‘next’ field indicates description of the next message field or contains ‘NULL’ in case of the last field. The ‘id’, ‘name’ and ‘desc’ fields specify identifier, name and description of the field, respectively. In case of different message schemes, these fields may contain null values. The ‘type’ field contains the name of the field type, which may be used to determine the operating mode of this field. The most common field types are the following:

i1, i2, i4, i8	Integer unsigned values with the size of 1, 2, 4 and 8 bytes, respectively
u1, u2, u4, u8	Integer unsigned values with the size of 1, 2, 4 and 8 bytes, respectively
cNN	String with the maximum length NN (ended with zero-value byte)
dMM.NN	Figure in the binary-decimal format with the total number of MM digits and the NN digits after the decimal point
bNN	Block with non-formatted binary data of the NN size
t	Structure describing date and time

The ‘size’ field contains the field value size and the ‘offset’ field — offset of this field in bytes from the beginning of the data block. This information provides unambiguous identification of the desired field location and size in the data block of the message.

The ‘def_value’ field contains indicator of the default value. Type and size of the value completely coincide with the type and size of the field, therefore initialization of the field by the default value may be performed simply by copying. The ‘NULL’ value of the ‘def_value’ field indicates absence of the default value.

The ‘values’ field indicates the first value of the allowed values list. The ‘NULL’ value of the ‘values’ field indicates that the field may take any value from the type definition domain.

```
struct cg_field_value_desc_t {
    // indicator of the next value
    struct cg_field_value_desc_t* next;

    // value name
    char* name;

    // value description
    char* desc;

    // indicator of the allowed value
    void* value;

    // for fields of the integer type (i[1-8], u[1-8]), the mask
    // determining the range of bytes taken by the value

    void* mask;
};
```

The ‘next’ field indicates the next value from the list of admissible field values or contains the NULL value, indicating the last value.

The ‘name’ and ‘desc’ fields contain name and description of the value.

The ‘value’ field indicates the field value; in this case, the size and type of the value coincide with the size and type of the field itself.

The ‘mask’ field is used for grouping of mutually exclusive values, in this case values with different masks may be combined.

Let’s suggest that we are dealing with subscription for receiving the Plaza-2 data stream with the following data scheme:

```
[dbscheme:FutTrade]
table=orders_log
table=heartbeat

[table:FutTrade:orders_log]
field=replID,i8
field=replRev,i8
field=replAct,i8
field=id_ord,i8
field=sess_id,i4
```

```
[table:FutTrade:heartbeat]
field=replID,i8
field=replRev,i8
field=replAct,i8
field=server_time,t
```

This format of schemes description in the form of ini-files is accepted in the Plaza-2 system.

This scheme describes two tables (two messages) with a certain set of fields in each of them. Let's suggest that we are desired in getting application numbers from the table 'orders_log' and server time synchronization events from the 'heartbeat' table — these values are contained in the 'id_ord' field of the 'orders_log' message and the 'server_time' field of the 'heartbeat' message, accordingly.

There are two ways to analyze the received data — static, with application of pre-set data structures, and dynamic, with calculation of offsets in the desired fields as of the moment of the scheme receiving.

The *static approach* is based on the fact that at the stage of development data schemes which will be further used are fixed for the desired streams. After that, for the data schemes, descriptions of C language structures are generated manually or automatically, e.g. using the 'schemetool' tool — these descriptions correspond to formats of binary data blocks for each of the received messages (for Java or .NET languages, a code, which analyzes binary blocks of messages is generated instead of structures). In course of operation data of the received message are displayed to the structure corresponding to the message type, and then the necessary data processing is performed.

On the one hand, this approach allows to simplify development process; on the other hand, it fixes a certain format of data schemes which will require another preparation of data structures or of the binary block analysis code. In case of changes in data schemes, the old structures may be no longer displayed to the new message formats; in some cases, it may lead to hard-to-detect errors.

Important

When pre-set structures are used for data displaying, it is always necessary to check compliance of expected structure formats with the actual ones. The basic checking consists in comparison of data blocks sizes with their corresponding pre-set structures.

Pre-set data structures or the binary block analysis code may be generated with application of the 'schemetool' utility.

It may look like the following:

```
// Description of structures generated with
// the 'schemetool' utility


#pragma pack(push, 4)
// Scheme "FutTrade" description

    struct orders_log
    {
        signed long long replID;
        signed long long replRev;
        signed long long replAct;
        signed long long id_ord;
        signed int sess_id;

    };
    const int orders_log_index = 0;

    struct heartbeat
    {
        signed long long replID;
        signed long long replRev;
        signed long long replAct;
        struct cg_time_t server_time;

    };
    const int heartbeat_index = 1;


#pragma pack(pop)

// in the subscription handler

case RTSCG_MSG_STREAM_DATA:
{
     cg_msg_streamdata_t* replmsg = (cg_msg_streamdata_t*)msg;
     if (replmsg->msg_index == orders_log_index)
     {
```

```
        orders_log* ordlog = (orders_log *)replmsg->data;
        printf ("Order ID = %lld\n", ordlog->id_ord);
    }
    else
    if (replmsg->msg_index == heartbeat_index)
    {
        heartbeat* hb = (heartbeat *)replmsg->data;
        printf ("Server time = %d:%d:%d.%d\n",
                hb->server_time.hour, hb->server_time.min,
                hb->server_time.sec, hb->server_time.ms);
    }
}
```

The *dynamic approach* suggests an absence of a clearly fixed data scheme, on the contrary — every time a data scheme is generated from the scheme source (for instance, from the replication server), and the user code analyzes it and performs search of desired messages and fields in these messages.

This approach enables to create a more universal system which will be able to overcome non-critical changes in data schemes; on the other hand, dynamic analysis of the scheme is more complicated in implementation.

The first step of this approach is to prepare information about desired fields — it is necessary to analyze the applied data stream scheme and to record numbers of desired messages and offsets of desired fields:

```
// variables which will contain information
// required for analysis of received data
size_t index_orders_log; // index of the ‘orders_log’ message in the scheme
size_t offset_id_ord; // offset of the ‘id_ord’ field in the block

size_t index_hearbeart; // index of the ‘heartbeat’ message in the scheme
size_t offset_server_time; // offset of the ‘server_time’ field in the block
```

This information is sufficient to identify the message type and to find the necessary field in the binary block at the moment of data receiving. These fields are filled-in in the following way:

```
cg_scheme_desc_t* scheme; // initialized description of the data scheme

size_t msgidx = 0;
for (cg_message_desc_t* msgdesc = schemedesc->messages;
     msgdesc; msgdesc = msgdesc->next, msgidx ++)
{
    size_t fieldindex = 0;
    if (strcmp(msgdesc->name, "orders_log") == 0)
    {
        index_orders_log = msgidx;
        for (cg_field_desc_t* fielddesc = msgdesc->fields;
             fielddesc; fielddesc = fielddesc->next, fieldidx ++)
            if (strcmp(fielddesc->name, "id_ord") == 0 &&
                strcmp(fielddesc->type, "i8") == 0)
                offset_id_ord = fieldidx;
    }
    if (strcmp(msgdesc->name, "heartbeat") == 0)
    {
        index_heartbeat = msgidx;
        for (cg_field_desc_t* fielddesc = msgdesc->fields;
             fielddesc; fielddesc = fielddesc->next, fieldidx ++)
            if (strcmp(fielddesc->name, "server_time") == 0 &&
                strcmp(fielddesc->type, "t") == 0)
                offset_server_time = fieldidx;
    }
}
```

The specified code is featured by a consistent search of all messages in the scheme and search of necessary fields for desired messages. This is accompanied by checking of field types for compliance with expectations.

The received data are processed in the following way:

```
// in the subscription handler

case RTSCG_MSG_STREAM_DATA:
{
     cg_msg_streamdata_t* replmsg = (cg_msg_streamdata_t*)msg;

     // coercion to char* to be able to add offset in bytes correctly
     char* data = (char*)replmsg->data;
     if (replmsg->msg_index == index_orders_log)
```

```
    {
        int64_t id_ord = *((int64_t*)(data + offset_id_ord));
        printf ("Order ID = %lld\n", id_ord);
    }
    else
    if (replmsg->msg_index == index_heartbeat)
    {
        cg_time_t *srvtime = (cg_time_t*)(data + offset_server_time);
        printf ("Server time = %d:%d:%d.%d\n",
                srvtime->hour, srvtime->min, srvtime->sec, srvtime->ms);
    }
}
```

This example will display the application identifier upon delivery of data on changes in the application status and also the server time at the moment of the corresponding message delivery.

This example demonstrates the following useful practices for code generation:

- Monitoring of data types during analysis of the scheme — provides correct diagnosis of errors in case of changes in the schemes
- Usage of numeric message identifiers instead of strings — has positive impact on capacity; thus, instead of a more expensive operation of strings comparison, it is possible to compare two numbers
- No data copying — it is not necessary to address each field by calling a special function; data are available directly in the message buffer
- Maintenance of data schemes evolution — code, which analyzes the scheme upon opening of the stream will be able to work with different data schemes, with no necessity to change hardwired identifiers and to perform recompilation.

Sending transactions and receiving replies

Sending FORTS transactions and receiving replies on their running is performed via the ‘Publisher’ and ‘Listener’ objects. The created ‘Publisher’ object is tied to the connection by calling the ‘cg_pub_new’ function, e.g. in the following way:

```
result = cg_pub_new(conn,
        "p2mq://FORTS_SRV;category=FORTS_MSG;"
        "name=PUB;scheme=|FILE|ini/forts_scheme_messages.ini|message ",
        &pub);
```

In this example, ‘pub’ is initialized by the ‘Publisher’ object which is set for sending of FORTS transactions according to the scheme, which is stored in the ‘ini’ sub-directory with the ‘forts_scheme_messages.ini’ file name and the ‘message’ scheme name via the ‘conn’ connection. ‘Publisher’ was assigned with the ‘PUB’ name which will be referenced by ‘Listener’.

Upon successful calling of the ‘cg_pub_new’ function, the object turns into initialized state but still remains inactive. Further work with the publisher is possible only upon calling of the ‘cg_pub_open’ function:

```
result = cg_pub_open(pub, 0);
```

Opening parameters for the ‘Publisher’ object are not provided at the moment; therefore, a null pointer is transferred as the second parameter.

When the publisher has been created and opened, you may create and send transactions. To create a transaction, you may use the ‘cg_pub_msgnew’ function.

```
result = cg_pub_msgnew(pub, CG_KEY_NAME, “FutAddOrder”, &msgptr);
```

In this case, a message will be created for placing the FORTS order (‘FutAddOrder’ transaction) by name, and its indicator will be recorded into the ‘msgptr’ variable. The ‘cg_pub_msgnew’ function may also be used to create messages by the number in the active scheme and by the identifier.

A message is represented as a pointer to the ‘cg_msg_data_t’ structure:

```
struct cg_msg_data_t
{
  uint32_t type;    // Тип сообщения = CG_MSG_P2REPL_DATA
  size_t data_size; // Data size
  void* data;       // Data indicator

  size_t msg_index; // Message number in the active scheme
  uint32_t msg_id;  // Unique message identifier
  const char* msg_name; // Message name in the scheme

  uint32_t user_id; // User’s number of the message
  const char* addr; // Destination address
  struct cg_msg_data_t* ref_msg; // Reference message (not used now)
};
```

The ‘data’ field of the structure indicates the memory buffer of corresponding capacity, which should be filled according to the active scheme. The simplest way to do this is to bring the pointer to the correct structure. For instance, in the following way:

```
ord = (struct AddOrder*)msgptr->data;
strcpy(ord->broker_code, "HB00");
```

Description of the structure from the scheme may be created via the ‘schemetool’ utility.

When the message has been created and filled in, it must be sent out by the ‘cg_pub_post’ function:

```
 result = cg_pub_post(pub, msgptr, CG_PUB_NEEDREPLY);
```

The flag CG_PUB_NEEDREPLY means that we want to receive replies to the corresponding ‘lsnreply’ listener.

When the message has been sent, it may be destroyed by means of the ‘cg_pub_msgfree’ function:

```
result = cg_pub_msgfree(pub, msgptr);
```

The publisher is closed by calling the ‘cg_pub_close’ function:

```
result = cg_pub_close(pub);
```

This is accompanied by disconnection of the publisher from the ‘connection’ object; the object itself remains in the initialized state and may be re-opened. The object is destroyed by calling the ‘cg_pub_destroy’ function:

```
result = cg_pub_destroy(pub);
```

After that, the ‘pub’ object is released, and further work with this object is impossible.

The listener for receiving replies to commands is created in the following way:

```
result = cg_lsn_new(conn, “p2mqreply://;ref=PUB”, replyCB, user_data, &lsnreply);
```

This call initializes the ‘lsnreply’ variable with a special listener object in order to receive replies to the messages, which were sent by the publisher. Communication between the listener and the publisher is accomplished by name; in this case, this name is ‘PUB’, the ‘ref=PUB’ parameter of the initialization string creates this communication. One listener may be referenced to one publisher. Names of corresponding pairs should be unique. Messages containing replies to transactions and also information on other events of the publisher will be delivered to the ‘replyCB’ function. Life cycle of this ‘Listener’ object does not differ somehow from the life cycle of the replication listener, which is reviewed in the corresponding section except for the fact that ‘replyCB’ does not receive messages of the replication system but receives simple single 'MQ' messages, described by the ‘cg_msg_data_t’ structure. The ‘cg_msg_data_t’ structure is referenced to the data described by the scheme of the corresponding publisher, and there is also the ‘CG_MSG_P2MQ_TIMEOUT’ notification, if the time interval for waiting for the message reply was exceeded.

The user’s reply handler may look like:

```
CG_RESULT ClientMessageCallback(cg_conn_t* conn, cg_listener_t* listener, struct cg_msg_t* msg, void* data)
{
  switch (msg->type)
  {
    case CG_MSG_DATA:
    {
    uint32_t* data = msg->data;
    printf("Client received reply [id:%d, data: %d, user-id: %d, name: %s]\n",
      ((struct cg_msg_data_t*)msg)->msg_id,
      *((uint32_t*)msg->data),
      ((struct cg_msg_data_t*)msg)->user_id,
      ((struct cg_msg_data_t*)msg)->msg_name);

    {
      struct scheme_desc_t* scheme;
      size_t bufSize;

      if (cg_lsn_getscheme(listener, &scheme) != CG_ERR_OK)
        scheme = 0;

      if (cg_msg_dump(msg, scheme, 0, &bufSize) == CG_ERR_BUFFERTOOSMALL)
      {
        char* buffer = malloc(bufSize+1);

        bufSize++;
        if (cg_msg_dump(msg, scheme, buffer, &bufSize) == CG_ERR_OK)
          printf("client dump: %s\n", buffer);
        free(buffer);

      }
    }
    break;
    }
  case CG_MSG_P2MQ_TIMEOUT:
```

```
  {
    printf("Client reply TIMEOUT\n");
    break;
  }
  default:
    printf("Message 0x%X\n", msg->type);
  }
  return CG_ERR_OK;
}
```

This user's handler either outputs the dump of messages by means of the auxiliary 'cg_msg_dump' function or, in case of the reply period exceeding, monitors this situation and displays the corresponding messages on the screen.

In order to link the sent messages and their replies, it is necessary to use the 'user_id' field of the 'cg_msg_data_t' structure: setting 'user_id' on the sent message provides receiving of the reply message with the same 'user_id'.

API description

General agreements

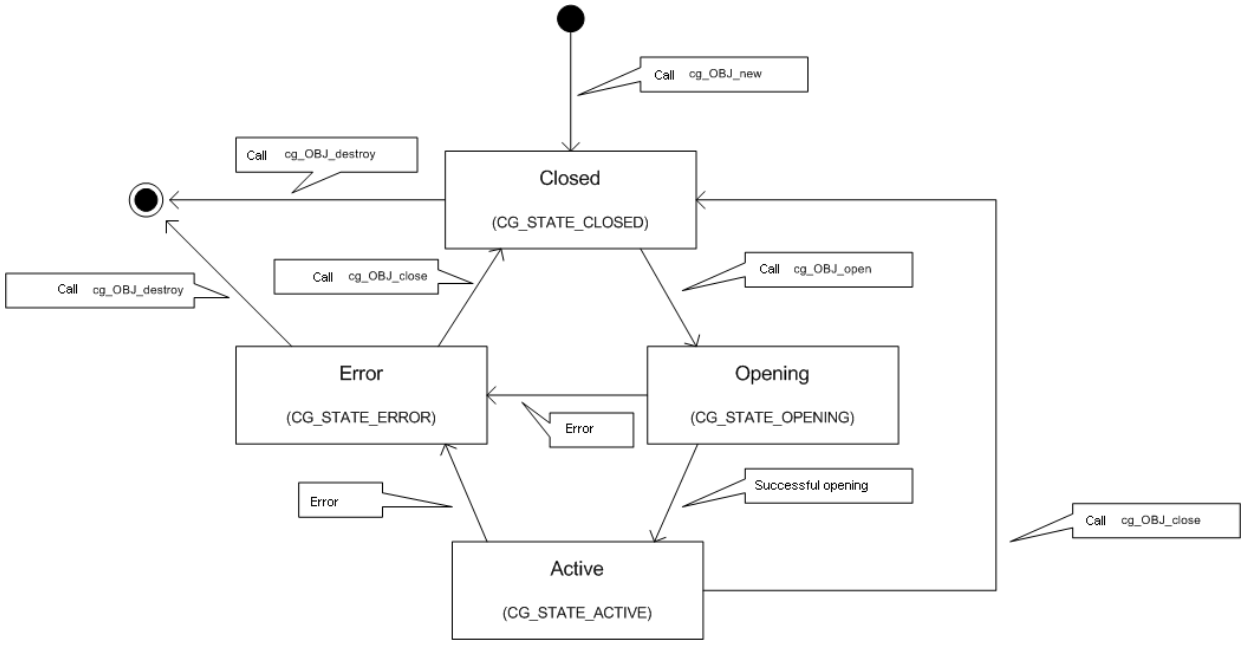
Program interface of the library is structured with allowance for a number of agreements:

- Each API function returns an error code
- Output parameters are set as indicators of variables, which should accept the value returned by the function and are located in the end of the parameters list
- Functions have prefixes, which usually consist of two parts: the first part "cg_" denotes the reference to the Client Gate library, the second part identifies the class of objects used with the respective function
  - env_ — functions that work with the general system environment
  - conn_ — functions that work with the connection
  - lsn_ — functions that work with the listeners
  - pub_ — functions that work with the publishers
  - log_ — functions that work with the operating log

  , also, there are certain functions which only have the 'cg_' prefix — these are the auxiliary functions and service functions, which do not belong to any particular group.
- Functions of the 'lsn_new', 'pub_new', etc. types create and initialize objects, which should be than released by corresponding calls of 'lsn_destroy', 'pub_destroy', etc. If objects are not destroyed explicitly, it will lead to memory leaks.

Life cycle of objects

The objects accessed through the library have the life cycle described by the following scheme:



Throughout their life cycle, objects exist in the following states:

- CG_STATE_CLOSED

  Closed state. An object is created in this state (upon calling ‘cg_OBJ_new’) and passes to it upon calling ‘cg_OBJ_close’.

- CG_STATE_OPENING

  This is an intermediate state between the closed and the active states. An object exists in this state upon ‘calling cg_OBJ_open’ and up to passing to the ‘CG_STATE_ACTIVE’ state or, in case of error in the object opening, to the ‘CG_STATE_ERROR’ state.

- CG_STATE_ACTIVE

  This is the active state — the main operating state of the object. In this state, an object may be dealt with — processing connection events, sending and receiving messages. An object passes to this state upon completion of the opening process from the state ‘CG_STATE_OPENING’. An object may pass from this state either to the state ‘CG_STATE_CLOSED’ upon calling of the function ‘cg_OBJ_close’, or to the state ‘CG_STATE_ERROR’ in case of error.

- CG_STATE_ERROR

  This is the error state. An object turns out to be in this state if an error occurs in the course of its opening or operation. An object may be passed from this state to the closed state by calling ‘cg_OBJ_close’ or destroyed by calling ‘cg_OBJ_destroy’, if further work with the object is not required.

This scheme of states is used for the following objects:

- Connections — cg_conn_t
- Listeners — cg_listener_t
- Publishers — cg_publisher_t

Usage in the multithreading environment

The ‘CGate’ library may be used in the multithreading environment *but it is not thread-safe*. It means that in order to provide correct work with the multithreading library, it is necessary to follow the certain rules:

- Work with the ‘Connection’ object should be performed only from one thread at any time.

  In this respect, the correct thing to do is to create a connection from one thread and to work with it from another thread. In this case, it is essential that several threads wouldn’t perform actions with the connection at the same time. If it is necessary to separate the connection between several threads at the same time, it is required to use primitive elements of synchronization from the operation system to provide synchronization of access to the ‘Connection’ object.

- Operations with the ‘Listener’ and ‘Publisher’ objects should be performed only from one thread at any time, in the same way as from the ‘Connection’ object.

- The ‘Listener’ and ‘Publisher’ objects are tied to a particular connection (the connection which was set at the moment of their creation) and work with these objects should be performed from the same thread which is used for work with the connection.

  Otherwise, it may lead to the situation when the user code will try to use the ‘Connection’ object at the same time when, for instance, the ‘Listener’ object uses the same connection. Thus, it will lead to incorrect operation of the library.

Connection

The ‘Connection’ object provides interaction with the Plaza-2 router for sending and receiving of messages. Any amount of these objects may be created at any time during program operation with initialized environment; nevertheless, it is recommended to create connections at the start of program and to stop them just before exiting.

cg_conn_new

Connection is created by calling:

```
CG_RESULT cg_conn_new(const char* settings, cg_conn_t** connptr);
```

Parameters are represented by the connection initialization string and by a pointer, which includes the pointer to the created connection. The connection creation string is set in the URL format in the following way: `"TYPE://HOST:PORT;param1=value1;param2=value;...;paramN=valueN"`, where

TYPE Connection type. Two connection types are currently supported:

p2tcp Connection to the Plaza-2 router via the TCP/IP protocol. It is slower, better for debugging, and may be linked with a router that is installed on third computer.

p2lrpcq Connection to the Plaza-2 router via shared memory. It is faster, better for production, works only with a single computer.

HOST Destination address of connection. In case of connection of the ‘p2tcp’ type — this is the address of the computer, where the desired P2MQRouter process was started, in case of ‘p2lrpcq’ — the address is 127.0.0.1.

PORT Number of the port which is used to create connection. It should be specified for both ‘p2tcp’ and ‘p2lrpcq’; in the latter case, the port will be used as a control channel for connection creation via shared memory.

Allowed parameters for adjustment of the ‘p2tcp’ and ‘p2lrpcq’ connections:

app_name Plaza-2 application name. Within one Plaza-2 router, each connection with the router should have a unique name. This identifier is used for routing of messages to the corresponding handlers.

local_pass Password for connection with the Plaza-2 router, if the router is configured to verify the connections to be opened.

timeout Time in milliseconds spent on waiting for connection creation with the router in the process of calling ‘conn_open(...)’. If this time exceeded, calling ‘conn_open(...)’ returns an error.

local_timeout Time in milliseconds spent on waiting for reply from the Plaza-2 router, when the ‘p2lrpcq’ connection is used.

Example of function call:

```
const char* conn_str = "p2lrpcq://127.0.0.1:4001;app_name=myapp";
cg_conn_t* conn;

result = cg_conn_new(conn_str, *conn);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to initialize connection: 0x%X\n", result);
    return;
}
```

cg_conn_open

Connection is opened by the call:

```
CG_RESULT cg_conn_open(cg_conn_t* conn, const char* settings);
```

Parameters are represented by the connection object indicator and the connection opening string. The connection opening string is not currently used and should be either empty or ‘NULL’.

Return values:

CG_ERR_OK Successful running.

CG_ERR_INVALIDARGUMENT Invalid arguments were transferred to the function.

CG_ERR_INCORRECTSTATE An attempt was made to open the connection, when it was impossible to open it, since it is either already active or is in the error state.

CG_ERR_INTERNAL Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see the analysis of the library logs.

Important

If the function returns the ‘CG_ERR_OK’ value, it doesn’t mean that the connection was successfully opened — this may be determined only on the basis of the connection status change (‘cg_conn_getstate’). Successful running of this function means that the connection opening process was successfully initiated, and some time later the connection may pass to the ‘CG_STATE_ACTIVE’ state in case of a succeeded connection opening, or to the ‘CG_STATE_ERROR’ state in case of a failure in connection opening.

Example of function call:

```
cg_conn_t* conn; // indicator of the object initialized by calling ‘conn_new’

result = cg_conn_open(conn, NULL);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to open connection: 0x%X\n", result);
    // Try to reopen the connection
}
```

cg_conn_close

Connection is closed by calling:

```
CG_RESULT cg_conn_close(cg_conn_t* conn);
```

The parameter is represented by the connection object pointer.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INCORRECTSTATE	An attempt was made to close the connection when it has been already closed.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

After the closure of the connection, it may be reopened by calling cg_conn_open.

Example of function call:

```
cg_conn_t* conn; // indicator of the object initialized by calling ‘conn_new’

result = cg_conn_close(conn);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to close connection: 0x%X\n", result);
    return;
}
```

cg_conn_destroy

Connection is destroyed by calling:

```
CG_RESULT cg_conn_destroy(cg_conn_t* conn);
```

The parameter is represented by the connection object pointer.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INCORRECTSTATE	An attempt was made to destroy the connection, when it has not been correctly closed.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

This call destroys the object which is indicated by the ‘conn’ parameter and releases all associated resources. After the function was called, the object cannot be used anymore. This function must be called for every object created by calling cg_conn_new, regardless of the fact if the tasks (opening, data acquisition, sending messages) were performed with this object or not.

Example of function call:

```
cg_conn_t* conn; // of the object which was closed by the calling conn_close

result = cg_conn_destroy(conn);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to destroy connection: 0x%X\n", result);
    return;
}
```

cg_conn_process

Messages of the connection are processed by calling:

```
CG_RESULT cg_conn_process(cg_conn_t* conn, uint32_t timeout, void* reserved);
```

Parameters are represented by the connection object indicator and the events timeout. The last parameter (‘reserved’) is not currently used and must be equal to ‘NULL’.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_TIMEOUT	No events of the system were processed in the specified time period

CG_ERR_INVALIDARGUMENTInvalid arguments were transferred to the function.

CG_ERR_INTERNAL Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

This calling performs iteration of work with the connection which includes examination of the queue of incoming messages, analysis of delivered data, calling of user-defined callback functions. This function must be called from the user code with the appropriate frequency corresponding to the maximum desired rate of data acquisition.

Important

User-defined callback functions for listeners linked to this connection will be called in the course of this function operation from the same running thread.

If the value of the ‘timeout’ parameter is not equal to 0, the calling will be blocked for ‘timeout’ milliseconds while waiting for events. If at the moment of the function calling the queue of incoming messages is not empty, the function will immediately proceed with processing of incoming messages.

Example of function calling:

```
cg_conn_t* conn; // indicator of the active connection

// analysis of incoming messages in the cycle, but no more than 100 per iteration
for (int callidx = 0; callidx < 100; ++ callidx)
{
    result = cg_conn_process(conn, 0, NULL);

    if (result == CG_ERR_TIMEOUT) // no messages
        break; // proceed with further logic of program operation
    else
    if (result != CG_ERR_OK)
    {
        // failure of the attempt to process the connection
        // display the message and close the connection
        fprintf(stderr, "Failed to process connection: 0x%X\n", result);
        result = cg_conn_close(conn); //
        if (result != CG_ERR_OK)
        {
            // failure to close the connection, exit the program
            fprintf(stderr, "Failed to close connection: 0x%X\n", result);
            return;
        }
        break;
    }
}
```

Listener

The ‘Listener’ object provides receiving of messages via the specified connection. Rules of messages receiving depend on the listener type — these may be both peer-to-peer messages and publish-subscribe messages, such as replication.

Operations with the ‘Listener’ objects in API are performed via the ‘cg_listener_t*’ indicator.

cg_lsn_new

Listener is created by calling:

`CG_RESULT cg_`**`lsn_new`**`(cg_conn_t* `*`conn`*`, const char* `*`settings`*`, CG_LISTENER_CB `*`callback`*`, void* `*`data`*`, cg_listener_t** `*`lsnptr`*`);`

Parameters are represented by: pointer the initialized connection object where a listener is created, listener initialization string, indicator of the callback function, that will be called upon certain events, arbitrary pointer that will be transferred to the callback function and a pointer that will include the pointer of the created listener.

The connection creation string is set in the URL format in the following way:

`"TYPE://[STREAM][;param1=value1[;param2=value[;...[;paramN=valueN]]]]"`, where

TYPE Subscription type. The following subscription types are available:

p2repl Receiving a table thread of the Plaza-2 replication

p2mqreply Receiving replies to previously sent messages

p2ordbook — Receiving active applications with usage of orderbooks snapshots for initial synchronization, and then switching to the online thread

The other parameters depend on the subscription type.

Parameters available by the *"p2repl"* subscription type:

STREAM — Sets the name of the table replication thread.

"scheme" parameter — Path to the applied data scheme of the thread. See Data schemes.

Parameters available by the '*p2ordbook*' subscription type:

STREAM — Specifies the name of the online thread with the orders operations log (FORTS_FUTTRADE_REPL or FORTS_OPTTRADE_REPL)

"snapshot" parameter — Name of the thread with snapshot of active orders (FORTS_FUTORDERBOOK_REPL or FORTS_OPTORDERBOOK_REPL).

"scheme" parameter — Path to the applied data scheme of online thread. Scheme must contain the 'orders_log' table.

"snapshot.scheme" parameter — Path to the applied data scheme of the snapshot thread. The scheme should contain the 'orders' and 'info' tables.

Parameters available by the '*p2mqreply*' subscription type:

"ref" parameter — Contains the name of the publisher which was used for sending of the messages with replies to be received in this listener

The 'p2mqreply' listener uses the scheme set in the associated publisher as a data scheme. This data scheme will be returned by calling cg_lsn_getscheme.

The 'p2ordbook' listener uses a combination of snapshots and online threads as a scheme. In this respect, data at the moment of snapshot delivery will correspond to messages from the scheme with data snapshots, and after switching to the online mode the messages from the online scheme will be delivered. When static data structures are used for working with the messages of this thread, descriptions of structures must be available both for snapshot and for online data, and in the respective attention should be paid to the indices of corresponding tables. When the dynamic approach is applied to work with schemes, standard practices must be used — remember numbers of desired messages and fields and use them upon data delivery.

The 'callback' parameter of the function indicates the user-defined callback function which looks like:

```
CG_RESULT callback(cg_conn_t* conn, cg_listener_t* listener, struct cg_msg_t* msg, void* data);
```

This function is called upon occurrence of any event on this subscription: opening of subscription, closing, message delivery, etc. Parameters of the callback function are represented by the indicator of the subscription connection, the indicator of the subscription object that is related to the event, the message indicator and the user 'data' indicator which was transferred for calling 'cg_lsn_new'. User handler return code should be set to 0 in case of successful processing of the message or to another value in case of error.

Important

Calling of the 'cg_lsn_new' function performs only initialization of the subscription object but does not lead to the actual commencement of data receiving; to commence with the receiving of data, you should switch the subscription to the active state by calling cg_lsn_open.

The user-defined callback function may include the following messages:

Subscriber type	Message type	Description
p2repl, p2mqreply	CG_MSG_OPEN	The message is delivered at the moment of data stream activation. This event surely occurs before receiving of any data on this subscription. For data streams, delivery of the message means that the data scheme was agreed and is ready for usage (For more details, see Data schemes). This message does not contain additional data, and its 'data' and 'data_size' fields are not used.
p2repl, p2mqreply	CG_MSG_CLOSE	The message is delivered at the moment of data stream closure. Delivery of the message means that the stream was closed by the user or by the system. This message does not contain additional data, and its 'data' and 'data_size' fields are not used.
p2repl	CG_MSG_TN_BEGIN	Indicates the moment when receiving of the next data block is started. It may be used by the program logics for control of data integrity together with the next message. This message does not contain additional data, and its 'data' and 'data_size' fields are not used.
p2repl	CG_MSG_TN_COMMIT	Indicates the moment when receiving of the next data block is completed. By the moment of this message delivery, it may be assumed that data received under this subscription are in the consistent state and reflect tables in the inter-synchronized state.

Subscriber type	Message type	Description
		This message does not contain additional data, and its ‘data’ and ‘data_size’ fields are not used.
p2repl	CG_MSG_STREAM_DATA	The message indicating delivery of stream data. The ‘data_size’ field contains the amount of data received; ‘data’ indicates the information itself. The message itself contains additional fields, which are described by the 'rtscg_msg_streamdata_t' structure. For more information see Receiving data streams
p2repl	CG_MSG_P2REPL_ONLINE	Stream switching to the online mode — it means that receiving of the initial snapshot has been completed, and the 'CG_MSG_P2REPL_DATA' messages bear online data. This message does not contain additional data, and its ‘data’ and ‘data_size’ fields are not used.
p2repl	CG_MSG_P2REPL_LIFENUM	The scheme life number was changed. This message means that previous data which were received on the stream are not urgent and should be cleaned. This will be accompanied by re-translation of data on the new data scheme life number. The ‘data’ field of the message indicates an integer value containing the new scheme life number; the ‘data_size’ field indicates the size of the integral type.
p2repl	CG_MSG_P2REPL_CLEARDELETED	Mass deletion of outdated data was performed. The ‘data’ field of the message indicates the ‘cg_data_cleardeleted_t’ structure which indicates the number of table and the number of revision — data in this table issued prior to this revision are deemed to be deleted.
p2repl	CG_MSG_P2REPL_REPLSTATE	The message indicates the state of data stream; it is sent before closure of the stream. The ‘data’ field of the message indicates the string which indicates the encoded state of the data stream as of the moment of the message delivery — the data scheme, table revision numbers and the scheme life number are preserved. This string may be transferred for calling the ‘cg_lsn_open’ function as the ‘replstate’ parameter on the same stream for the next time; therefore, data receiving will continue upon shutdown of the stream.
p2mqreply	CG_MSG_DATA	The message contains a reply to the previously sent state. The ‘data’ field indicates data, and the ‘data_size’ field contains the size of the data block. The message is described by the ‘cg_msg_data_t’ structure and contains additional fields allowing to identify the initial message along with information about the data scheme. For more details see Sending commands and receiving replies.
p2mqreply	CG_MSG_P2MQ_TIMEOUT	p2mqreply CG_MSG_P2MQ_TIMEOUT The message is delivered if the reply to the previously sent message was not received during the time period specified in the corresponding publisher. The message is described by the ‘cg_msg_data_t’ structure and contains the ‘user_id’ value which is set upon sending of the initial message.

Example of the listener created for receiving of the data stream:

```
cg_conn_t* conn; // indicator of the ‘Connection' initialized object

const char* lsn_str = "p2repl://FORTS_FUTINFO_REPL";
cg_listener_t* lsn;

result = cg_lsn_new(conn, lsn_str, callback, 0, *lsn);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to initialize listener: 0x%X\n", result);
    return;
}
```

Example of the listener created for receiving of replies to commands sent:

```
cg_conn_t* conn; // indicator of the ‘Connection' initialized object

// command sending publisher initialization string
// the ‘name=TN1’ parameter is specified

const char* pub_str = "p2mq://FORTS_SRV;category=FORTS_MSG;name=TN1";
cg_publisher_t* pub;

// reply receiving listener initialization string
// the ‘ref=TN1’ parameter is specified, it provides communication with the publisher
const char* lsn_str = "p2mqreply://;ref=TN1";
cg_listener_t* lsn;

result = cg_lsn_new(conn, lsn_str, callback, 0, *lsn);
if (result != CG_ERR_OK)
```

```
{
    fprintf(stderr, "Failed to initialize listener: 0x%X\n", result);
    return;
}
```

cg_lsn_open

Subscription is opened by calling:

```
CG_RESULT cg_lsn_open(cg_listener_t* lsn, const char* settings);
```

Parameters are represented by the subscription object indicator and the subscription opening string. The opening parameters string is set in the "param1=value1;param2=value2;....;paramN=valueN" format, and in this case the names and values of parameters depend on the type of subscription.

Parameters of the '*p2repl*' subscription opening are the following:

mode — Specifies the data receiving mode and may take the following values:

- snapshot — The thread is opened in the data snapshot receiving mode. In this case, online data will not be transmitted
- online — The thread is opened in the online data receiving mode. Data snapshot will not be received, data will be transferred upon opening of the stream
- snapshot+online — The thread is opened in the snapshot receiving mode, and then passes to the online data receiving mode.

replstate — Specifies the stream state which should be used for opening. Value of this parameter must correspond to the string received in the 'CG_MSG_P2REPL_REPLSTATE' message upon previous closure of the stream.

lifenum — Sets the scheme life number. This parameter may be used for connection to the data stream if the possibilities provided by the ‘replstate’ parameter are not suitable by any reason. If the ‘replstate’ parameter is set, then the value of this parameter will be ignored.

rev.TABLE_NAME — Sets the initial revision of the table TABLE_NAME. The name of the desired table should be inserted instead of TABLE_NAME. This parameter may be used for connection to the data stream if the possibilities provided by the 'replstate' parameter are not suitable by any reason. If the ‘replstate’ parameter is set, then the value of this parameter will be ignored. This parameter may be specified several times for different tables in the stream, e.g. ‘rev.orders_log=234445;rev.deal=55’.

Return values:

CG_ERR_OK — Successful running.

CG_ERR_INVALIDARGUMENT — Invalid arguments were transferred to the function.

CG_ERR_INCORRECTSTATE — An attempt was made to open the subscription when it was impossible to open it since it is either already active or in the error state.

CG_ERR_INTERNAL — Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

Important

If the function returns the ‘CG_ERR_OK’ value, it does not mean that the subscription was successfully opened — this may be specified only by the subscription status change (‘cg_lsn_getstate’). Successful running of this function indicates that the subscription opening process was successfully started, and some time later the subscription may switch to the ‘CG_STATE_ACTIVE’ state in case of success, or to the ‘CG_STATE_ERROR’ state in case of a failure at opening.

Example of function calling:

```
cg_listener_t* lsn; // indicator of the object initialized by calling ‘cg_lsn_new’
const char* lsn_open_str = "mode=online";

result = cg_lsn_open(lsn, lsn_open_str);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to open listener: 0x%X\n", result);
    // Try to reopen the listener
}
```

cg_lsn_close

Subscription is closed by calling:

```
CG_RESULT cg_lsn_close(cg_listener_t* lsn);
```

The parameter is represented by the subscription object indicator.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INCORRECTSTATE	An attempt was made to close the connection when it has been closed.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

After closing of the subscription, it may be reopened by calling cg_lsn_open.

Example of function calling:

```
cg_listener_t* lsn; // pointer to the opened listener

result = cg_lsn_close(lsn);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to close listener: 0x%X\n", result);
    return;
}
```

cg_lsn_destroy

Subscription is destroyed by calling:

```
CG_RESULT cg_lsn_destroy(cg_listener_t* lsn);
```

The parameter is represented by the subscription object indicator.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INCORRECTSTATE	An attempt was made to destroy the connection, when it has not been correctly closed.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

This calling destroys the object which is indicated by the ‘lsn’ parameter and releases all associated resources. Upon calling of this function, the object cannot be used any more. This function must be called for every object created by calling ‘cg_lsn_new’, regardless of the fact whether the actions (opening, data acquisition, sending of messages) were or were not performed with this object.

Example of function calling:

```
cg_listener_t* lsn; // indicator of the object which was closed by calling cg_lsn_close

result = cg_lsn_destroy(lsn);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to destroy listener: 0x%X\n", result);
    return;
}
```

cg_lsn_getstate

Listener status is received by calling:

```
CG_RESULT cg_lsn_getstate(cg_listener_t* lsn, uint32_t* state);
```

Parameters are represented by the subscription object indicator and the value indicator with the size of 4 bytes which will include the current status of the listener.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.

CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

This calling may be used for periodic receiving of the listener status to be able to perform actions associated with switching the subscription object to different states – for instance, to close subscription if it switched to the error state. For more details on the object status, see the Life cycle of objects section.

Example of function calling:

```
cg_listener_t* lsn; // Pointer to the listener object
uint32_t state; // Here status will be written

result = cg_lsn_getstate(lsn, &state);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to query listener state: 0x%X\n", result);
    return;
}

switch (state)
{
    case CG_STATE_ERROR: /* ... */
    case CG_STATE_CLOSED: /* ... */
}
```

cg_lsn_getscheme

Listener's scheme is received by calling:

```
CG_RESULT cg_lsn_getscheme(cg_listener_t* lsn, cg_scheme_desc_t** schemeptr);
```

Parameters are represented by the subscription object pointer and the pointer of the variable which will include the scheme description pointer.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see the analysis of the library logs.

Calling is used to receive the data scheme of the subscription object (for more details see Work with data schemes). The data scheme is available upon delivery of the 'OPEN' event for subscription. If the data scheme was not explicitly set at the moment of subscription creation, the scheme may be changed between two working sessions, i.e. in the general case you cannot rely on the situation when, in the chain of open/close calls, the open/close scheme after the first open will be the same as the scheme after the second open call. This may be decided either by indicating the client data scheme, when it is supported by the subscription type or by analyzing the scheme each time when the 'OPEN' event is delivered.

Example of function calling:

```
cg_listener_t* lsn; // subscription object indicator
cg_scheme_desc_t* schemedesc; // Scheme description indicator will be recorded here

result = cg_lsn_getscheme(lsn, &schemedesc);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to query listener scheme: 0x%X\n", result);
    return;
}

// print the number of messages in the scheme
printf("Number of messages: %d\n", schemedesc->num_messages);
```

Publisher

The 'Publisher' object provides sending of messages via the specified connection. Rules of messaging depend on the publisher type and connection.

Operations with the 'Publisher' objects in API are performed by using the 'cg_publisher_t*' pointer.

cg_pub_new

Listener is created by calling:

```
CG_RESULT cg_pub_new(cg_conn_t* conn, const char* settings, cg_publisher_t** pubptr);
```

Parameters are represented by: a pointer to the initialized connection object where a publisher is created, publisher initialization string and a pointer which will include the pointer of the created listener.

The connection creation string is set in the URL format in the following way: "TYPE://[NAME][;param1=value1[;param2=value[;...[;paramN=valueN]]]]",

where

TYPE	Publisher type. The following types are supported: p2mq Sending arbitrary messages to Plaza-2
"name" parameter	Defines a unique name of the publisher. May be used for interaction between paired publishers and listeners (for instance, the ‘mq’ publisher and ‘mqreply’ listener).

The other parameters depend on the subscription type.

Parameters available by the *‘p2mq' publisher* type:

NAME	Specifies the name of the service which will receive messages sent by this publisher.
"scheme" parameter	Path to the applied data scheme (see Data schemes). The applied data scheme must contain descriptions of requests and replies — the associated ‘p2mqreply’ listener will use this data scheme in the analysis of messages.
"category" parameter	Category of messages being sent. To send commands to the FORTS trading system, this parameter should be fixed as ‘FORTS_MSG’
"timeout" parameter	Time spent on waiting for the reply to the sent message in milliseconds.

.

Important

Calling of the ‘cg_pub_new’ function only initializes the publisher object, but does not actually allow to send messages; in order to commence messaging, you should send the publisher to the active state by calling cg_pub_open.

Example of the listener created for data sending to the trading system:

```
cg_conn_t* conn; // pointer to the initialized Connection object

const char* pub_str = "p2mq://FORTS_SRV;category=FORTS_MSG;name=TN1";
cg_publusher_t* pub;

result = cg_pub_new(conn, pub_str, *pub);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to initialize publisher: 0x%X\n", result);
    return;
}
```

For the further information on receiving reports to the publisher commands, see the section describing the cg_lsn_new function.

cg_pub_open

Publisher is opened by calling:

```
CG_RESULT cg_pub_open(cg_publisher_t* pub, const char* settings);
```

Parameters are represented by the publisher object indicator and the opening string. At the present moment, publishers do not require setting of the parameter string, and this parameter should be ‘NULL’ or there should be empty string.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INCORRECTSTATE	An attempt was made to open the publisher when it was impossible to open it since it was either already active or in the error state. Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

Important

If the function returns the ‘CG_ERR_OK’ value, it does not mean that the publisher was successfully opened — this may be determined only by the subscription status change (cg_pub_getstate). Successful running of this function means that the publisher opening process was successfully started, and some time later the subscription may pass to the ‘CG_STATE_ACTIVE’ state in case of a success, or to the ‘CG_STATE_ERROR’ state in case of a failure in opening.

Example of function call:

```
cg_publisher_t* pub; // indicator of the object initialized by calling ‘cg_pub_new’

result = cg_pub_open(pub, 0);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to open publisher: 0x%X\n", result);
    // Try to reopen the publisher
}
```

cg_pub_close

Publisher is closed by calling:

```
CG_RESULT cg_pub_close(cg_publisher_t* pub);
```

The parameter is represented by the publisher object pointer.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INCORRECTSTATE	An attempt was made to close the connection when it is closed.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

After closure of the listener, it may be reopened by calling cg_pub_open.

Example of function call:

```
cg_publisher_t* pub; // indicator of the opened publisher

result = cg_pub_close(pub);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to close publisher: 0x%X\n", result);
    return;
}
```

cg_pub_destroy

Publisher is destroyed by calling:

```
CG_RESULT cg_pub_destroy(cg_publisher_t* pub);
```

The parameter is represented by the publisher object pointer.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INCORRECTSTATE	An attempt was made to destroy the connection when it wasn’t correctly closed.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

This calling destroys the object which is indicated by the ‘pub’ parameter and releases all associated resources. Upon calling of this function, the object cannot be used any more. This function must be called for every object created by the calling cg_pub_new ‘’, regardless of the fact if actions (opening, sending of messages) were performed with this object or not.

Example of function call:

```
cg_publisher_t* pub; // indicator of the object which was closed by calling cg_pub_close

result = cg_pub_destroy(pub);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to destroy publisher: 0x%X\n", result);
    return;
}
```

cg_pub_getstate

Publisher status is received by calling:

`CG_RESULT` **`cg_lsn_getstate`**`(cg_listener_t* lsn, uint32_t* state);`

Parameters are represented by the publisher object pointer and the value pointer with the size of 4 bytes which will include the current status of the publisher.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

This calling may be used for periodic receiving of the publisher status to be able to perform actions associated with switching of the subscription object to different states – for instance, to close the publisher if it switched to the error state. For more details on the object status, see the ‘Life cycle of objects’ section.

This function is available for calling at any time between calls of cg_pub_new and cg_pub_destroy.

Example of function call:

```
cg_publisher_t* pub; // publisher object indicator
uint32_t state; // Status will be recorded here

result = cg_pub_getstate(pub, &state);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to query publisher state: 0x%X\n", result);
    return;
}

switch (state)
{
    case CG_STATE_ERROR: /* ... */
    case CG_STATE_CLOSED: /* ... */
}
```

cg_pub_getscheme

Publisher’s scheme is received by calling:

`CG_RESULT` **`cg_pub_getscheme`**`(cg_publisher_t* pub, cg_scheme_desc_t** schemeptr);`

Parameters are represented by the publisher object pointer and the pointer to the variable which will include the scheme description pointer.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see the analysis of the library logs.

Calling is used to get the data scheme of the subscription object (for details see Work with data schemes). The data scheme is available upon the publisher switching into the ‘ACTIVE’ mode. If the data scheme was not explicitly set at the moment of publisher creation, the scheme may be changed between two working sessions, i.e. in general case you cannot rely on the situation when, in the chain of open/close calls, the open/close

scheme after the first open will be the same as the scheme after the second open call. This may be decided either by indication of the client data scheme when it is supported by the subscription type, or by analysis of the scheme each time when the ‘OPEN’ event is delivered.

Example of function call:

```
cg_publisher_t* pub; // publisher object indicator
cg_scheme_desc_t* schemedesc; // Scheme description indicator will be recorded here

result = cg_pub_getscheme(pub, &schemedesc);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to query publisher scheme: 0x%X\n", result);
    return;
}

// print number of messages in scheme
printf("Number of messages: %d\n", schemedesc->num_messages);
```

cg_pub_msgnew

A new message for sending is created by calling:

`CG_RESULT` **`cg_pub_msgnew`**`(cg_publisher_t*` *`pub`*`, uint32_t` *`id_type`*`, const void*` *`id`*`, struct cg_msg_t**` *`msgptr`*`);`

Parameters are represented by the publisher object pointer, the message key type, the key value pointer and the pointer of the variable which will include the pointer to the created message.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

This calling initializes a message for sending via this publisher. The desirable message is identified by the key type and value in the publisher data scheme. The following key types are available:

CG_KEY_INDEX	The key is represented by the message number in the scheme. The ‘id’ parameter indicates the value of the ‘uint32_t’ type, which stores the desirable message number
CG_KEY_ID	The key is represented by the unique numeric identifier of the message in the scheme. The ‘id’ parameter indicates the value of the ‘uint32_t’ type which stores the desirable message identifier
CG_KEY_NAME	The key is represented by the message name in the scheme. The ‘id’ parameter indicates the string, which includes the name of the desirable message. The string should end with null.

A message created by this function is the message of the ‘CG_MSG_DATA’ type and is described by the extended structure:

```
struct cg_msg_data_t
{
    // Message type. Always CG_MSG_DATA for this message
    uint32_t type;
    // Amount of data
    size_t data_size;
    // Pointer to data
    void* data;

    // Message description number in the active scheme
    size_t msg_index;
    // Unique identifier of the message type
    uint32_t msg_id;
    // Message name in the active scheme
    const char* msg_name;

    // User ID of the message
    uint32_t user_id;
    // Address of the opposite party
    const char* addr;
    // Reference message indicator
    struct cg_msg_data_t* ref_msg;
};
```

The ‘data_size’ field contains the size of the selected memory block for the requested message format, and the ‘data’ field points to this memory block. The ‘msg_index’, ‘msg_id’ and ‘msg_name’ fields are filled with data according to the applied data scheme. The ‘user_id’ field may be used for setting the user ID of the message — the same ‘user_id’ will be indicated in the reply message, which allows to bind the request and the reply.

The user code should compare the block size in the ‘data_size’ field, which is detached for the message with the expected version regarding the size of this block in order to avoid errors while filling the message. Then it is required to fill in the block using the ‘data’ indicator. After that, the message is ready for sending.

Example of function call:

```
cg_publisher_t* pub; // publisher object indicator
cg_msg_data* msg;

result = cg_pub_msgnew(pub, CG_KEY_NAME, "FutDelOrder", &msg);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to allocate message: 0x%X\n", result);
}
else
{
    FutDelOrder* delord;
    if (msg->data_size != sizeof(*delord))
    {
        fprintf(stderr, "Block sizes do not match: %d expected, but got %d \n",
                sizeof(*delord), msg->data_size);
    }
    else
    {
        delord = (FutDelOrder*)msg->data;
        delord->order_id = ...; // number of the deleting application

        result = cg_pub_post(pub, msg, CG_PUB_NEEDREPLY);
        if (result != CG_ERR_OK)
        {
            fprintf(stderr, "Failed to post message: 0x%X\n", result);
        }
    }
}
```

cg_pub_post

Message is sent by calling:

```
CG_RESULT cg_pub_post(cg_publisher_t* pub, struct cg_msg_t* msg, uint32_t flags);
```

Parameters are represented by the publisher object indicator, the message indicator and message sending flags.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INCORRECTSTATE	An attempt was made to send a message when the connection is not active.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

The calling tries to send the message. The message to be sent should be preliminarily initialized by calling ‘cg_pub_msgnew’and filled with user’s data. The ‘CG_PUB_NEEDREPLY’ value may be indicated as flags which notifies the system on the necessity to wait for the reply to the sent message.

Reply message may be received by means of the subscription of the ‘p2mqreply’ type (for more details see description of the cg_lsn_new function).

Example of function call:

```
cg_publisher_t* pub; // publisher object indicator
cg_msg_data* msg; // initialized message indicator

result = cg_pub_post(pub, msg, CG_PUB_NEEDREPLY);
if (result != CG_ERR_OK)
{
```

```
    fprintf(stderr, "Failed to post message: 0x%X\n", result);
}
cg_pub_msgfree(pub, msg);
```

cg_pub_msgfree

Message is released by calling:

`CG_RESULT` **`cg_pub_msgfree`**`(cg_publisher_t*` *`pub`*`, struct cg_msg_t*` *`msg`*`);`

Parameters are represented by the publisher object indicator and the indicator of the message to be released.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_INTERNAL	Internal error. May indicate a malfunction of configuration or running environment. For more detailed diagnosis, see analysis of the library logs.

The calling destroys the previously selected message. After this function has been called, the message, the ‘msg’ parameter points to, becomes unavailable for further usage, and all resources associated with this message are released. The function must be called for any message created by the ‘cg_pub_msgnew’ function, when the message has been sent and work with this message has been completed.

Example of function call:

```
cg_publisher_t* pub; // publisher object indicator
cg_msg_data* msg; // initialized message indicator

result = cg_pub_msgfree(pub, msg);
if (result != CG_ERR_OK)
{
    fprintf(stderr, "Failed to post message: 0x%X\n", result);
}
```

Auxiliary functions

cg_bcd_get

The function allows to receive a BCD-number in the form of two components: the integral part and the decimal point position.

`CG_RESULT` **`cg_bcd_get`**`(void*` *`bcd`*`, int64_t*` *`intpart`*`, int8_t*` *`scale`*`);`

Parameters of the function are represented by the number pointer in the BCD-format (‘bcd’), the pointer of the variable which will include the number value in the form of integral number and the pointer of the variable which will include the decimal point position in relation to the end of the number.

For instance, for the initial number 123.45 the function will record the value 12345 into the ‘intpart’ variable, and the value 2 into the ‘scale’ variable.

Important

The maximum number of signs represented in the form of a 64-bit integral number, is equal to 19. In order to receive values of BCD-numbers with the size exceeding 19 signs, you should use the 'cg_getstr' call to represent numbers in the form of strings.

Return values:

CG_ERR_OK	Successful running.
CG_ERR_INVALIDARGUMENT	Invalid arguments were transferred to the function.
CG_ERR_OVERFLOW	The delivered number is too large to be represented as 64-bit integer number.

Пример вызова функции:

```
void* bcd; // indicator of the BCD-number
int64_t value; // the integral number will be recorded here
int8_t scale; // the decimal point position will be recorded here

result = cg_bcd_get(bcd, &value, &scale);
if (result != CG_ERR_OK)
{
```

```
    fprintf(stderr, "Failed to convert decimal: 0x%X\n", result);
}

// print the value as a floating-point number
printf("Value is: %f\n", (double)value/pow(10.0, scale));
```

cg_getstr

The function allows to receive an arbitrary type string representation.

CG_RESULT **cg_getstr**(char* *type*, void* *data*, char* *buffer*, size_t* *buffer_size*);

Parameters of the function are represented by the field type in the Plaza-2 format (see the 'Work with data schemes' section) as the ‘type’ string, pointer to memory area, which stores the ‘data’ value, pointer to the buffer, which will include the ‘buffer’ string representation and the pointer to a variable that contains the buffer size value (‘buffer_size’).

If the buffer size is too small for recording of the string representation, the function will return the 'CG_ERR_BUFFERTOOSMALL' error code and will record the required buffer size in the ‘buffer_size’ field.

Return values:

CG_ERR_OK Successful running.

CG_ERR_INVALIDARGUMENTInvalid arguments were transferred to the function.

CG_ERR_BUFFERTOOSMALLThe transferred buffer is too small for string representation of the type.

Пример вызова функции:

```
void* bcd; // indicator of the BCD-number

char buf[32];
size_t bufsize = sizeof(buf);

result = cg_getstr("d26.2", bcd, buf, &bufsize);
if (result == CG_ERR_BUFFERTOOSMALL)
{
    char* buf2 = new char[bufsize];
    result = cg_getstr("d26.2", bcd, buf2, &bufsize);
    if (result != CG_ERR_OK)
        fprintf(stderr, "Failed to convert value: 0x%X\n", result);
    else
        printf("Value is %s\n", buf2);
    delete[] buf2;
}
else
if (result != CG_ERR_OK)
{
    printf("Value is %s\n", buf2);
}
else
{
    fprintf(stderr, "Failed to convert value: 0x%X\n", result);
}
```

cg_msg_dump

The function allows to get a text dump of an arbitrary message.

CG_RESULT **cg_msg_dump**(struct cg_msg_t* *msg*, struct cg_scheme_desc_t* *schemedesc*, char* *buffer*, size_t* *buffer_size*);

Parameters of the function are represented by the message pointer (‘msg’), the scheme description pointer (‘schemedesc’), pointer to the buffer which will include the text dump ‘buffer’ and pointer to the variable which contains the buffer size value (‘buffer_size’).

If the buffer size is too small for recording of the string representation, the function will return the ‘CG_ERR_BUFFERTOOSMALL’ error code and will record the required buffer size value in the ‘buffer_size’ field.

Return values:

CG_ERR_OK Successful running.

CG_ERR_INVALIDARGUMENTInvalid arguments were transferred to the function.

CG_ERR_BUFFERTOOSMALL The delivered buffer is too small for message dump.

If the ‘schemedesc’ parameter is not equal to ‘NULL’, the function will try to analyze the message using the transferred scheme. If the ‘schemedesc’ parameter is equal to ‘NULL’ or there is no message in the scheme or the message size doesn’t coincide with the size specified in the scheme, the function will output a hexadecimal dump of the message.

It is convenient to use this function for debugging.

Example of function calling:

```
cg_msg_t* msg; // message indicator
size_t bufsize = 0;

result = cg_msg_dump(msg, 0, 0, &bufsize);
if (result == CG_ERR_BUFFERTOOSMALL)
{
    char* buf = new char[bufsize];
    result = cg_msg_dump(msg, 0, buf, &bufsize);
    if (result != CG_ERR_OK)
    {
        fprintf(stderr, "Failed to dump message: 0x%X\n", result);
    }
    else
    {
        printf("%s\n", buf);
    }
    delete[] buf;
}
else
    fprintf(stderr, "Failed to dump message: 0x%X\n", result);
```

Tools description

‘Schemetool’ utility

The ‘schemetool’ utility is designed for working with data schemes.

At the present moment the system supports the function of data structures formation on programming languages corresponding to the format of data stream messages.

makesrc - structures generation

The ‘makesrc’ mode is used to form the initial code with the description of the message structures. The generated structures may be used for access to message fields.

A scheme is generated by the following call:

```
schemetool makesrc [options] [SOURCE SCHEME]
```

, where:

SOURCE Scheme source. Description of the scheme may be obtained from the ini-file, in this case the ini-file path should be transferred as ‘SOURCE’. Scheme may be also generated from the data stream — in this case, two parameters must be transferred as ‘SCHEME_SOURCE’: ‘--conn CONN_STR’ and ‘--stream STREAM_NAME’; in this case, ‘CONN_STR’ sets the string of connection with the ‘P2MQRouter’ router in the URL format, and ‘STREAM_NAME’ sets the name of the desired data stream.

SCHEME The desired scheme name, must be specified explicitly.

and 'options' are represented by the following parameters:

-o, --output `-o FILENAME`

Output file name. Results of the utility operation are recorded into this file. If this parameter is not specified, then the default output ‘stdout’ are used.

-O, --output-format `--output-format FORMAT`

Structure description format. At the moment, the following formats are supported:

- c — C langiage structures
- java — Java language classes

	• cs — C# language classes • pas — Pascal language structures
-Dgen-table-prefix=1	Used for the ‘c’ format. Usage of this key will lead to the situation when names of message structures will be added with prefixes — names of message schemes. This mode may be used to avoid name conflicts when using several schemes in the same INI-file in the situation when messages with the same names exist in different schemes.
-Dgen-namespaces=1	Used for the ‘c’ format. Usage of this key will lead to formation of ‘namespace’ with the scheme name for each scheme of the INI-file. May be used to resolve the name conflict as an alternative to the previous option if the C++ compiler is used for program compilation.
-Djava-class-name=CLASSNAME	Used for the 'java' format. Allows to specify the generating upper level Java class name.
-Djava-user-package=PACKAGE	Used for the 'java' format. Allows to specify the packet name of the generating Java class.
-Djava-time-format=date	Used for the 'java' format. The fields containing the 'date-time' type values will be converted into 'java.util.Date' (by default).
-Djava-time-format=long	Used for the 'java' format. The fields containing the 'date-time' type values will be converted to a 'long' type values, containing amount of milliseconds passed from 00:00:00 01.01.1970.
-Djava-bcd-format=bigdecimal	Used for the 'java' format. The fields containing the BCD-type values will be converted into 'java.math.BigDecimal' (by default)
-Djava-bcd-format=long	Used for the 'java' format. The fields containing the BCD-type values will be converted into 'long'.
-Dnet-user-namespace=NAMESPACE	Used for the 'cs' format. Allows to specify the .NET name space for the generating class.
-Dnet-time-format=datetime	Used for the 'cs' format. The fields containing the 'date-tyme' type values will be converted into 'DateTime' (by default).
-Dnet-time-format=long	Used for the 'cs' format. The fields containing the 'date-time' type values will be converted to a 'long' type values, containing amount of milliseconds passed from 00:00:00 01.01.1970.
-Dnet-bcd-format=decimal	Used for the 'cs' format. The fields containing the BCD-type values will be converted into 'decimal' (by default).
-Dnet-bcd-format=long	Used for the 'cs' format. The fields containing the BCD-type values will be converted into 'long'.

Examples of the utility usage:

```
schemetool makesrc -o futinfo.h forts_scheme.ini FUTINFO
```

- this example forms descriptions of structures in the ‘futinfo.h’ file using the C language for the FUTINFO scheme from the ‘forts_scheme.ini’ file.

```
schemetool makesrc -o futinfo.pas --output-format pas forts_scheme.ini FUTINFO
```

- this example forms descriptions of structures in the ‘futinfo.h’ file using the Pascal language for the ‘FUTINFO’ scheme from the ‘forts_scheme.ini’ file.

```
schemetool makesrc -o futinfo.h --output-format c \
                   --conn p2tcp://localhost:4001;app_name=stool \
                   --stream FORTS_FUTINFO_REPL
```

- this example forms descriptions of structures in the ‘futinfo.h’ file using the C language for the data scheme of the ‘FORTS_FUTINFO_REPL’ stream accessible via the connection with the Plaza-2 router which is enabled on the same computer over the port 4001.

```
schemetool makesrc -o messages.h forts_messages.ini message
```

- this example forms descriptions of structures in the ‘messages.h’ file for messages of the FORTS trading system from the ‘forts_messages.ini’ file.

API for Java and .NET description

Description

The following interface libraries are included into the P2 CGate distribution kit:

- cgate_java

  The library implementing the interface with Java platform

- cgate_net

  The library implementing the interface with .NET platform

Both libraries are developed according to similar rules and operate similar objects. Due to that, API description will be referred simultaneously for Java and .NET.

API CGate for Java

CGate support for Java is implemented through JNI interface. P2 CGate supply package includes the following components, related to Java support:

- cgate jni interface library (cgate_jni.dll for Windows, libcgate_jni.so for Linux; CGATE_HOME/p2bin) catalogue
- Java cgate.jar class library (CGATE_HOME/sdk/lib catalogue)
- Example of using P2 CGate in Java (catalogue CGATE_HOME/sdk/samples/java)

The following are necessary for using P2 CGate in Java:

- use the 'cgate.jar' library during the project compilation
- while launching the project:
  - have 'cgate.jar' in classpath
  - have the 'cgate_jni' library in the path, used for loading the dynamic libraries (specified with 'Java.library.path')
  - have a set of P2 CGate libraries available for loading (CGATE_HOME/p2bin catalogue content)

It is possible to explicitly define the used cgate_jni library and paths to it using the following properties:

- ru.micexrts.cgate.name

  Specifies the library file name

- ru.micexrts.cgate.path

  Specifies the folder with the library file

For example:

```
java -cp .;lib/cgate.jar -Dru.micexrts.cgate.name=libcgate_jni.so.1 -Dru.micexrts.cgate.path=. MyApp
```

In this example, a user app of the 'MyApp' class is launched, using the 'cgate.jar' library from the ‘lib’ sub-catalogue; the 'cgate_jni' interface library will be taken from the './libcgate_jni.so.1' file.

API CGate for .NET

Support of CGate for .NET platform is implemented through C++/CLI. P2 CGate distribution kit includes the following components, referring to the .NET support:

- cgate_net.dll build(CGATE_HOME/p2bin) catalogue)
- Example of using P2 CGate in.NET (CGATE_HOME/sdk/samples/java catalogue)

In order to use P2 CGate from .NET it is necessary:

- to use the 'cgate_net.dll' build during the project compilation
- while launching the project :
  - have 'cgate_net.dll' available for loading by .NET platform
  - have a set of P2 CGate libraries available for loading (CGATE_HOME/p2bin catalogue content)

Important

Launching of the 'cgate_net' library at Mono platform is not supported.

'Connection' object

The ‘Connection’ object provides access for the connection function (see Connection).

Description	Java	.NET	CGate API function
Creation of connection object	Connection(String settings)	Connection(string settings)	cg_conn_new
Destruction of connection object	void dispose()	void Dispose()	cg_conn_destroy
Opening connection	void open(String settings)	void Open(string settings)	cg_conn_open
Closing connection	void close()	void Close()	cg_conn_close

Description	Java	.NET	CGate API function
Processing connection messages	void process(int timeout)	void Process(int timeout)	cg_conn_process
Receiving connection status	int getState()	State	cg_conn_getstate

Important

Upon the connection termination the dispose() method must be called, which evidently releases the resources, related to the connection.

Connection constructor

Initializes the new class instance.

Java syntax:

```
public Connection(String settings) throws CGateException
```

C# syntax:

```
public Connection(string settings)
```

where:

settings Connection initialization string (see cg_conn_new)

Possible exceptions:

CGateException Connection creation error

'Connection.dispose' method

Connection resources cleanup is performed by calling the 'dispose()' method.

Java syntax:

```
public void dispose() throws CGateException
```

C# syntax:

```
public void Dispose()
```

Possible exceptions:

CGateException Connection destruction error

'Connection.open' method

Connection is opened by calling the 'open()' method.

Java syntax:

```
public void open(String settings) throws CGateException
```

C# syntax:

```
public void Open(string settings)
```

Possible exceptions:

CGateException Connection opening error

Connection.close method

Connection is opened by calling the 'close()' method.

Java syntax:

```
public void close() throws CGateException
```

C# syntax:

```
public void close()
```

Possible exceptions::

CGateException Connection closing error

Connection.process method

Connection messages are processed by calling the 'process()' method.

Java syntax:

```
public int process(int timeout)
```

C# syntax:

```
public int Process(int timeout)
```

Possible exceptions: none.

Return values:

CG_ERR_OK	Operation completed successfully
CG_ERR_INVALIDSTATE	Invalid connection state
CG_ERR_INTERNAL	Internal error

'Connection.state' property

Connection messages are processed by calling the 'process()' method.

Java syntax:

```
public int getState() throws CGateException
```

C# syntax:

```
public State State { get; }
```

Possible exceptions:

CGateException	Connection closing error

Return values:

CLOSED	Connection is closed
ERROR	Connection is in the error state
OPENING	Connection is being opened
ACTIVE	Connection is active

Listener object

The Listener object provides access to the set of listener functions (see Listener).

Description	Java	.NET	CGate API function
Creation of the listener object	Listener(Connection conn, String settings, ISubscriber subscriber)	Listener(Connection conn, string settings)	cg_lsn_new
Destruction of the listener object	void dispose()	void Dispose()	cg_lsn_destroy
Opening the listener	void open(String settings)	void Open(string settings)	cg_lsn_open
Closing the listener	void close()	void Close()	cg_lsn_close
Receiving connection status	int getState()	State	cg_lsn_getstate
Receiving listener scheme	int getScheme()	Scheme	cg_lsn_getscheme
Handler installation	-	Handler	-

Important

When the work with subscriber is completed, the dispose() method is to be called, which evidently released the related resources.

Listener constructor

Initializes the new class instance.

Java syntax:

```
public Listener(Connection conn, String settings, ISubscriber subscriber) throws CGateException
```

C# syntax:

```
public Listener(Connection conn, string settings)
```

where:

conn connection, which the listeneris linked to

settings listener initialization string (see cg_lsn_new)

subscriber user message handler (only Java; in case of .NET, the 'Handler' properties should be used)

Possible exceptions:

CGateException subscription creation error

for Java the subscriber parameter indicates the class instance, which implements the 'ISubscriber' interface.

```
public interface ISubscriber {

    public int onMessage(Connection conn, Listener listener, Message message);
}
```

In case of generation of some listener event, for example, arrival of new message or change in the listener state, the object's 'onMessage' method passed as a parameter subscriber will be called.

The following parameters will be passed:

conn Connection to which a listener is bound

listener The listener, in which the event occured

msg Message

The 'subscriber' parameter is absent for .NET; An extra 'Handler' property is introduced instead of it, which allows installing the message handler in the most natural way for .NET environment.

'Listener.dispose' method

Listener resources cleanup is performed through calling the 'dispose()' method.

Java syntax:

```
public void dispose() throws CGateException
```

C# syntax:

```
public void Dispose()
```

Possible exceptions:

CGateException Listener destruction error

'Listener.open' method

Attempts to open the listener.

Java syntax:

```
public void open(String settings) throws CGateException
```

C# syntax:

```
public void Open(string settings)
```

Possible exceptions:

CGateException Listener opening error

'Listener.close' method

Closes the subscription

Java syntax:

```
public void close() throws CGateException
```

C# syntax:

```
public void close()
```

Possible exceptions:

CGateException Listener closing error

'Listener.State' ptoperty

Returns the current listener state.

Java syntax:

```
public int getState() throws CGateException
```

C# syntax:

```
public State State { get; }
```

Possible exceptions:

CGateException Subscriber status acquisition error

Return values:

CLOSED Listener is closed

ERROR Listener is in the error state.

OPENING Listener is being opened

ACTIVE Listeber is active

'Listener.Scheme' property

Returns the current listener data scheme

Java syntax:

```
public Scheme getScheme() throws CGateException
```

C# syntax:

```
public Scheme Scheme { get; }
```

Possible exceptions:

CGateException Data scheme obtaining error

Returned value is the description of the current listener data scheme, or null, if the listener operates without it.

Important

Listener data scheme is available since the ‘OPEN’ message is received and until the subscriber is closed or switched to the error mode.

Important

Listener data scheme may change between the two ‘CLOSE’ and ‘OPEN’ events; i.e. after the repeated opening of the listener, its scheme may differ from the one, that was effective during the recent activity session.

Listener.Handler property

Allows to install ther user message handler of the listener.

C# syntax:

```
public MessageHandler Handler { get; set; }
```

User handlers must comply with the following type:

```
delegate int MessageHandler(Connection conn, Listener listener, Message msg);
```

, where the following will be sent as parameters:

conn Connection, which the subscriber is linked to

listener The listener, where the event occurred

msg Message

Publisher object

The publisher object provides access to the publisher set of functions (see Publisher).

Description	Java	.NET	CGate API function
Creation of the publisher object	Publisher(Connection conn, String settings)	Publisher(Connection conn, string settings)	cg_pub_new
Destruction of the publisher object	void dispose()	void Dispose()	cg_pub_destroy
Opening of publisher	void open(String settings)	void Open(string settings)	cg_pub_open
Closing of publisher	void close()	void Close()	cg_pub_close
Publisher state acquisition	int getState()	State	cg_pub_getstate
Publisher scheme acquisition	int getScheme()	Scheme	cg_pub_getscheme
Creating a message for sending	Message newMessage(int idType, Object id)	Message NewMessage(MessageFlag idType, Object id);	cg_pub_msgnew
Sending a message	void post(Message msg, int flags)	Message Post(Message msg, PublisherFlag flags);	cg_pub_post

Important

When all operations with the publisher are completed, the 'dispose()' method is to be called, which explicitly releases the related resources.

Publisher constructor

Initializes the new class instance.

Java syntax:

```
public Publisher(Connection conn, String settings) throws CGateException
```

C# syntax:

```
public Publisher(Connection conn, string settings)
```

where:

conn Connection, which the publisher is linked to

settings Publisher initialization string (see cg_pub_new)

Possible exceptions:

CGateException Publisher creation error

'Publisher.dispose' method

Publisher resources cleanup is performed through calling the 'dispose()' method.

Java syntax:

```
public void dispose() throws CGateException
```

C# syntax:

```
public void Dispose()
```

Possible errors:

CGateException Publisher destruction error

Publisher.open method

Attempts to open the publisher.

Java syntax:

```
public void open(String settings) throws CGateException
```

C# syntax:

```
public void Open(string settings)
```

Possible errors:

CGateException Publisher opening error

Publisher.close method

Closes the publisher

Java syntax:

```
public void close() throws CGateException
```

C# syntax:

```
public void close()
```

Possible exceptions:

CGateException Publisher closing error

Publisher.State property

Returns the current publisher state.

Java syntax:

```
public int getState() throws CGateException
```

C# syntax:

```
public State State { get; }
```

Possible exceptions:

CGateException Publisher status acquisition error

Return values:

CLOSED Publisher is closed

ERROR Publisher is in the state of error

OPENING Publisher is being opened

ACTIVE Publisher is active

Publisher.Scheme property

Returns the current publisher data scheme.

Java syntax:

```
public Scheme getScheme() throws CGateException
```

C# syntax:

```
public Scheme Scheme { get; }
```

Possible exceptions:

CGateException Data scheme acquisition error

Returned value is the description of the current publisher data scheme, or null, if the publisher operates without it.

Important

Publisher data scheme may change between the ‘CLOSE’ and ‘OPEN’ events; i.e. after the repeated publisher opening, its scheme may differ from the one, that was effective during the recent activity session.

Publisher.newMessage method

Creates new message for sending.

Java syntax:

```
public void newMessage(int idType, Object id) throws CGateException
```

C# syntax:

```
public void NewMessage(MessageFlag idType, Object id)
```

, where:

idType Message identifier type. May take one of the following values:

- KEY_INDEX — id parameter is the unique number of the required message in the scheme
- KEY_ID — id parameter is the unique numerical identifier of the required message in the scheme
- KEY_NAME — id parameter is the name-string of the required message in the scheme

id Message identifier(Integer or String, depending upon the 'idType' pararameter value)

Possible exceptions:

CGateException Message creation error

The created message contains a buffer, which size fits the message description in the scheme.

Publisher.post method

Sends the message.

Java syntax:

```
public void post(Message msg, int flags) throws CGateException
```

C# syntax:

```
public void Post(Message msg, PublisherFlag flags)
```

, where:

msg Messages for sending

flags Message sending flags. At the moment only the 'NEED_REPLY' flag is supported, which means the nessecity to receive a reply to the sent message.

Possible exceptions:

CGateException Message sending error

Sent message is not used after calling the 'post()' method and can be deleted or used for re-sending.

Important

Publisher object may only send the messages, created by the same object instance.

Message object

The Message pbject provides access to messages.

Описание	**Java**	**.NET**	**CGate API**
Destruction of the message object	void dispose()	void Dispose()	cg_pub_msgfree
Acquisition of the message type	int getType()	Type	type field of the cg_msg_t structure
Acquisition of the data buffer	ByteBuffer getData()	Data	Data field of the cg_msg_t structure
Acquisition of the debugging message view	String toString()	string ToString()	cg_msg_dump

User is responsible for erasing the messages that were created for sending through explicit calling the 'dispose()' method. Messages that user receives to the subscription handler should not be deleted, since such messages are owned by P2 CGate library.

Метод Message.dispose

Message resources cleanup is performed through calling the 'dispose()' method.

Java syntax:

```
public void dispose() throws CGateException
```

C# syntax:

```
public void Dispose()
```

Possible exceptions:

CGateException Message destruction error

Message.Type property

Returns message type.

Java syntax:

```
public int getType()
```

C# syntax:

```
public MessageType Type { get; }
```

Message.Data property

Return the message data buffer.

Java syntax:

```
public java.nio.ByteBuffer getData()
```

C# syntax:

```
public System.IO.UnmanagedMemoryStream Data { get; }
```

Data buffer size is available through the respective object call, returned with the property. Buffer format fits the used data scheme.

Data property may return null — it means that the message does not contain data.

Message.toString method

Returns the text representation of the message.

Java syntax:

```
public String toString()
```

C# syntax:

```
public string ToString()
```

This representation may also be used for debugging purposes.

Message types

For the more comfortable work with P2 CGate, some classes were introduced, which describe the particular message types. Such classes contain additional information. User may gain access to additional message properties, having converted the object type (cust.), based upon the analysis of 'Message.Type' properties.

OpenMessage object

Describes the messages of CG_msg_OPEN type — opening of the listener.

Object does not contain additional fields.

CloseMessage object

Describes messages of CG_msg_CLOSE type — closing of the listener.

Object does not contain additional fields.

DataMessage object

Describes the messages of 'CG_msg_DATA' type — data message.

Additional object properties:

Description	Java	.NET	CGate API
Message number in the data scheme	int getMsgIndex()	MsgIndex	the msg_index field of the cg_msg_data_t structure
Numerical message identifier in the data scheme	int getMsgId()	MsgId	the msg_id field of the cg_msg_data_t structure
Message name in the data scheme	int getMsgName()	MsgName	the msg_name field od the cg_msg_data_t structure
Message sender/recipient address	string getAddress()	Address	the addr field of the cg_msg_data_t structure
User message number	int getUserId()/void setUserId(int val)	MsgName	the user_id field of the cg_msg_data_t structure
List of the message fields	Value[] getFields()	Fields	-
Field receiving according to the name	Value] getField(String name)	Field[string]	-

StreamDataMessage object

Describes the message type ‘CG_msg_STREAM_DATA’ — message stream data.

Additional object properties:

Description	Java	.NET	CGate API
Message number in the data scheme	int getMsgIndex()	MsgIndex	the msg_index field of the cg_msg_streamdata_t structure
Numerical message identifier in the data scheme	int getMsgId()	MsgId	the msg_id field of the cg_msg_streamdata_t structure
Message name in the data scheme	int getMsgName()	MsgName	the msg_name field of the cg_msg_streamdata_t structure
Message number in the stream	long getRev()	Rev	the rev field of the cg_msg_streamdata_t structure
List of the message fields	Value[] getFields()	Fields	-
Field receiving according to the name	Value] getField(String name)	Field[string]	-

TnBeginMessage object

Describes messages of the ‘CG_msg_TN_BEGIN’ type — identifies the transaction start for stream data.

Object does not contain additional fields.

TnCommitMessage object

Describes messages of the ‘CG_msg_TN_COMMIT’ type — identifying the transaction end for the stream data.

Object does not contain additional fields.

P2MQTimeoutMessage object

Describes messages of the ‘CG_msg_P2MQ_TIMEOUT’ type — notifying that the answer latency limit for the sent message exceeded.

Additional object properties:

Description	Java	.NET	CGate API
User message number	int getUserId()/void setUserId(int val)	MsgName	the user_id field of the cg_msg_data_t structure_t

P2ReplLifeNumMessage object

Describes messages of the ‘CG_msg_P2REPL_LIFENUM’ type — message notifying on the changed living data scheme number.

Additional object properties:

Description	Java	.NET	CGate API
New data scheme life number	int getLifeNumber()	LifeNumber	message *data value

P2ReplClearDeletedMessage object

Describes messages of the ‘CG_msg_P2REPL_CLEARDELETED’ type — data range deletion message for the specified message.

Additional object properties:

Description	Java	.NET	CGate API
Table number	int getTableIdx()	TableIdx	the table_idx field of the cg_data_cleardeleted_t structure
Revision number, data below which is deleted	long getTableRev()	TableRev	the table_rev field of the cg_data_cleardeleted_t structure

P2ReplOnlineMessage object

Описывает сообщений типа CG_MSG_P2REPL_ONLINE - сообщение о переходе потока данных в состояние ONLINE.

The object does not contain additional fields.

P2ReplStateMessage object

Describes messages of the ‘CG_msg_P2REPL_REPLSTATE’ type — message, containing the data stream state for re-opening.

Additional object properties:

Description	Java	.NET	CGate API
Stream re-opening data	String getReplState()	ReplState	message *data value