AccessLib C++ API

AGTEK Development Inc.
Research and Development
August 23, 2011

Table of Contents

1 Document History

Author	Date	Description
Michael Matthews	02/11/10	Initial version
Michael Matthews	02/16/10	Updated with Jonathan's edits
Michael Matthews	03/12/10	Added documentation of the RTK API
Michael Matthews	03/29/10	Added documentation of setPassword() and sendEmail()
Michael Allison	08/23/11	Add LicenseAPI information

2 Intended Audience

This document contains AGTEK proprietary information and is intended for use by AGTEK employees only.

3 Introduction

The AGTEK Access Server provides networked storage for AGTEK customer project data including GPS devices, project definitions, vehicle lists, vehicle-device associations, GPS track data and shared file storage. The AccessLib C++ API is a library of functions and objects designed for communicating with the AGTEK Access Server. The library hides the underlying TCP/IP implementation providing client applications with an easy to use high level API.

3.1 Third Party Dependencies

3.1.1 Google Protocol Buffer

The AGTEK Access Server uses the Google Protocol Buffer (protobuf) software for encoding client-server messages. The Google Protocol Buffer software allows application developers to describe messages in a simple platform independent syntax. These messages can then be compiled to generate code for several different platforms and computer languages. This increases flexibility by allowing client and server applications to be written in different languages and for different platforms. The AGTEK Access Server is written in Java running on a Linux server whereas some of the client applications are written in C++ running under Windows. Additionally, the messages are encoded in a compact binary form that makes efficient use of memory. All the communication between the AGTEK Access Server and the client applications use protobuf encoded messages. More information about the Google Protocol Buffer software can be found at: http://code.google.com/p/protobuf/.

3.1.2 Googletest

The source code for the AccessLib C++ library includes a set of unit tests based on the Googletest (gtest) framework. Googletest is a framework for writing C++ unit tests that compile and run on a variety of platforms. The framework includes support for automatic test discovery and a rich set of assertions. More information about gtest can be found at: http://code.google.com/p/googletest/.

3.2 Cloud Hosting

The AGTEK Access Server runs off-site utilizing the Amazon AWS cloud computing infrastructure. This relieves AGTEK of the need to run it's own network accessible data center. The server itself is running on a single “small EC2 instance” running Linux. The server requires a MySQL database which is running on a single “small RDS instance” completely managed by Amazon. File storage is handled by Amazon's S3 storage system. Finally, GPS tracks are stored in flat files on an Amazon EBS block device. All systems can be scaled according to need and have reliability exceeding our own existing data center solution. Current costs for the entire system are running less than the cost of the lab T1 alone and costs are trending downwards. More information about Amazon AWS cloud services (including explanations of all the above acronyms) can be found at: http://aws.amazon.com.

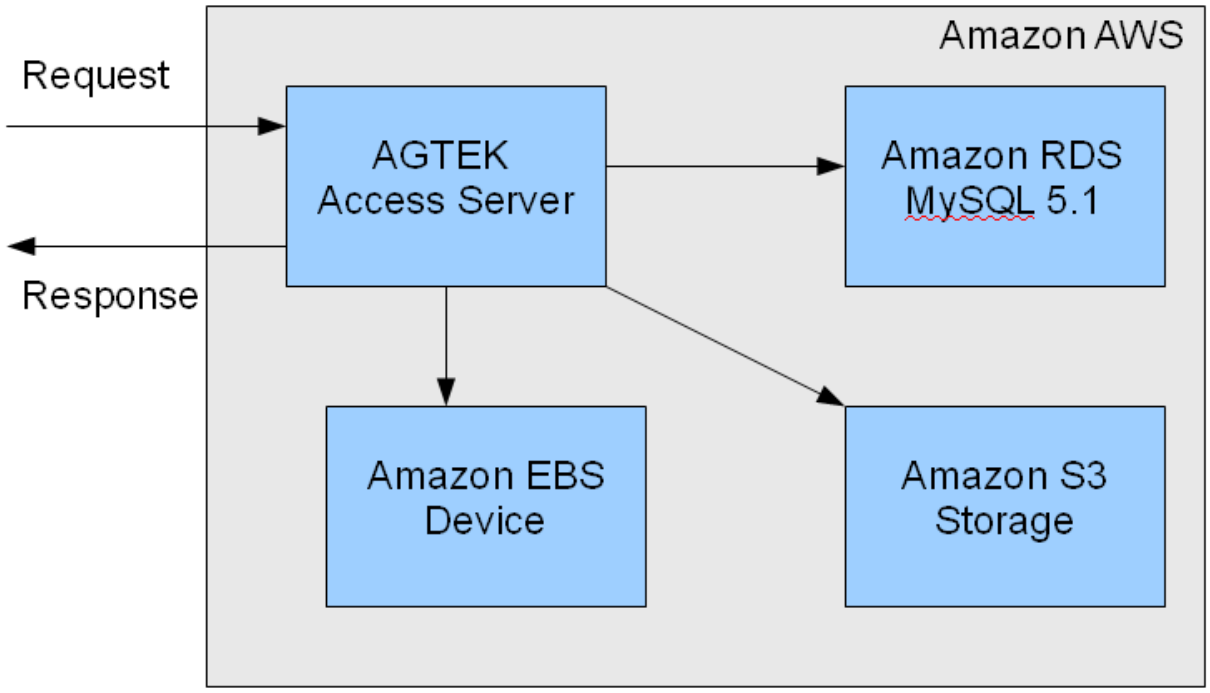

Illustration 1: AGTEK Access Server Diagram

4 Client-Server Connections

Using the AccessLib C++ API, client applications create connections to the AGTEK Access Server. Connections to the server are session based and re-useable. This allows multiple requests to be sent across a single connection. When all requests are complete, the client application disconnects which closes the connection and frees up any server side resources in use. The server will timeout and close connections that are inactive for more than a very short time (timeout is currently set to one minute).

Connections are **NOT** thread or process safe. Client applications that wish to perform requests across multiple threads or processes must create a new connection for each thread or process.

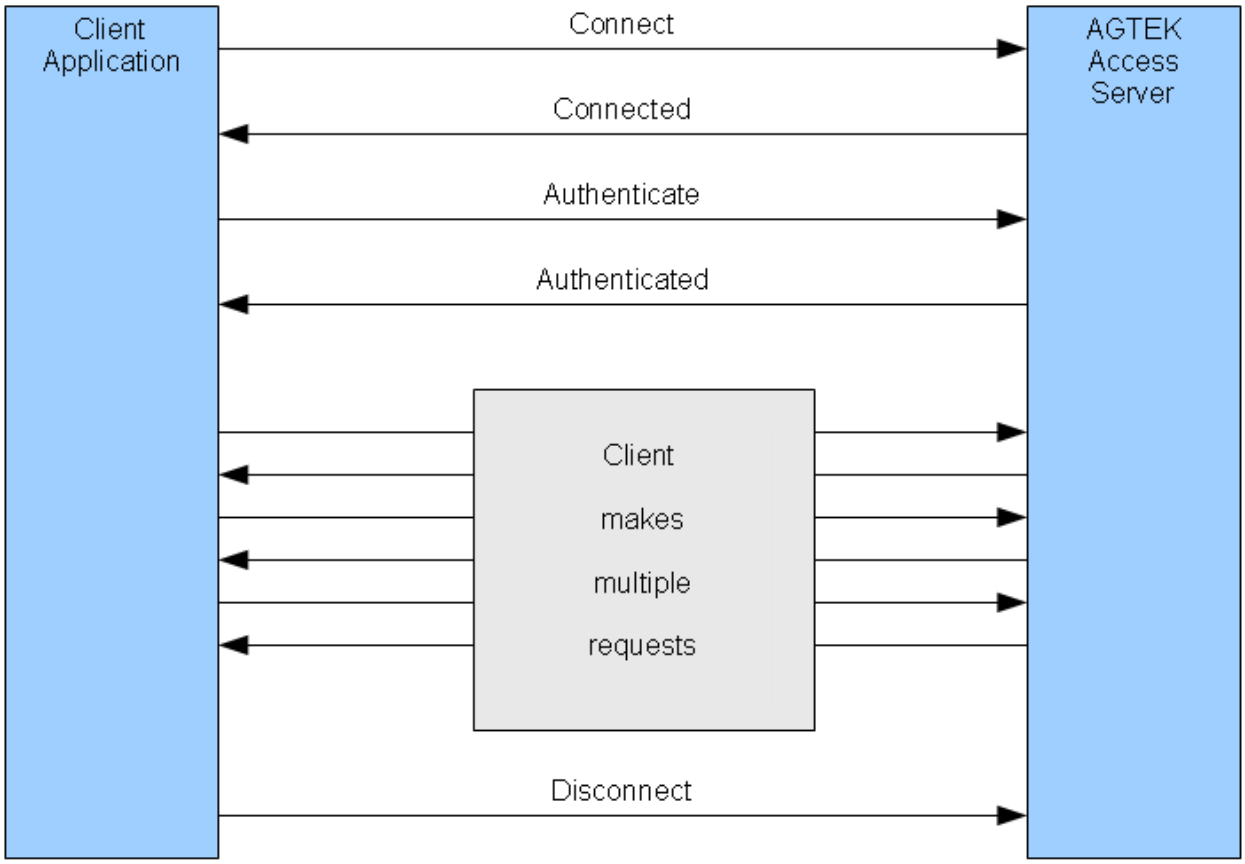

Illustration 2: Client-Server Communication Diagram

4.1 Authentication

A client application that is uploading GPS data for a device that has been registered on the server and assigned to a customer does not need to authenticate. For all other operations, the client application must authenticate using a valid user id and the user's password or authentication token. User ids and passwords are generated by either AGTEK support or by a designated administrator at the customer's site.

4.1.1 Connecting Without Authentication

Client applications can connect to the server and upload GPS data without authenticating. In order for the GPS data to be stored with the appropriate customer, the serial number of the GPS must be known and registered with the server. Additionally, the device must be assigned to the customer. Serial numbers are usually assigned by AGTEK support when a device is sold. All other client-server operations will fail on an unauthenticated connection.

4.1.1.1 Example Connecting Without Authentication

```
#include "access/AccessLib.h"

int connectExample(void) {
  // Initialize AccessLib
  // Do this only ONCE at application startup
  AccessError error = AccessLibInitialize();
  if (error) {
    printf("AccessLib init failed with error %d\n", error);
  }

  // Connect to the server
  IaccessServer* server = NULL;
  if (!error) {
    server = getAccessServer("access.agtek.com", 34015);
    error = server->connect();
  }

  // Print the server name and version
  if (!error) {
    printf("Connected to %s\n", server->name().c_str());

    // The client can upload GPS positions for registered devices now

    // All done, we should disconnect
    server->disconnect();
  }

  // Do this once, when the client application shutsdown
  AccessLibShutdown();

  return error;
}
```

4.1.2 Connecting With Authentication

All client applications that need to do more than simply upload GPS data for known devices must authenticate immediately after connecting to the server. Authentication requires a user id and either the user password, or an authentication token. An authentication token is a unique string returned after a successful authentication. The token represents a one-way encryption of the user password (not reversible) and is safe to cache on the client machine. The password and the authentication token are not interchangeable. The client application must call either the *authPassword()* or *authToken()* method as appropriate.

4.1.2.1 Example Connecting with Authentication

```
#include "access/AccessLib.h"

static std::string USER_ID("testuser"); // Example user id
static std::string PASSWORD("test");    // Example password
static std::string TOKEN;

int authenticateExample(void) {
  // Initialize AccessLib
  // Do this only ONCE at application startup
  AccessError error = AccessLibInitialize();
  if (error) {
    printf("AccessLib init failed with error %d\n", error);
  }

  // Connect to the server
  IaccessServer* server = NULL;
  if (!error) {
    server = getAccessServer("access.agtek.com", 34015);
    error = server->connect();
  }

  // Print the server name and version then authenticate
  if (!error) {
    printf("Connected to %s\n", server->name().c_str());

    // Check to see if we have a cached token
    if (TOKEN.length() > 0) {
      // Try using the cached token
      error = server->authToken(USER_ID, TOKEN);

    } else {
      // No cached token, send the user's password
      error = server->authPassword(USER, PASSWORD);

      if (!error) {
        // Print and save the token.
        // The token is safe to cache on the client machine.
        TOKEN = server->token();
        printf("Authentication token %s\n", TOKEN.c_str());
      }
    }
  }

  // All done, we should disconnect
  if (server)
    server->disconnect();

  // Do this once, when the client application shutsdown
  AccessLibShutdown();

  return error;
}
```

5 AccessLib API

The AccessLib API is divided into a small set of entry functions for initializing the library, shutting down the library and getting an instance of a server connection and a set of pure virtual C++ interface classes for manipulating customer data and managing the server connection.

5.1 Building AccessLib

The source code for the AccessLib C++ API is checked into Visual Source Safe under the "AccessLib" project. The project has three main components:

1. The AccessLib C++ library
2. A set of unit tests based on the Googletest framework
3. Example code

The AccessLib C++ library is the only component required by client applications and should be built first. The unit test should be built next to verify that the library is functioning correctly. Finally, if desired, the example code can be built and run.

5.1.1 Building Under Windows

Checked into source code control are Visual C++ project files for each of the three AccessLib components. They should be built in the following order:

1. AccessLib\AccessLib.vcproj
2. AccessLib\test\AccessLibTest.vcproj
3. AccessLib\examples\AccessLibExamples.vcproj

Notes:

- To run the unit tests it may be necessary to modify the file AccessLib\test\TestConfig.txt if the server, port, test user id or test password have changed.
- To run the examples it may be necessary to modify the HOST, PORT, USER and PASS constants defined at the top of the file AccessLib\examples\AccessLibExamples.cpp if the server, port, test user id or test password have changed.

5.1.2 Building Under Fedora/Ubuntu

Checked into source code control are makefiles for each platform, one under AccessLib/fedora and one under AccessLib/ubuntu. These makefiles build all three AccessLib components.

Notes:

- To run the unit tests it may be necessary to modify the file TestConfig.txt in the platform build directory if the server, port, test user id or test password have changed.
- To run the examples it may be necessary to modify the HOST, PORT, USER and PASS constants defined at the top of the file AccessLib\examples\AccessLibExamples.cpp if the

server, port, test user id or test password have changed.

5.2 AccessError Error Codes

All AccessLib functions and methods that communicate with the server return an AccessError enumerated error code. The value ACCESS_OK (integer value 0) denotes a successful call, no error. Any error code with a value greater than 2000 indicates a fatal error. Fatal errors leave the connection in an unknown state and the associated IAccessServer instance should be deleted by the client application and not reused. A comprehensive list of all the AccessError values can be found in Appendix A at the end of this document.

5.3 Entry functions

5.3.1 AccessLibInitialize()

The AccessLibInitialize function is called only once during client application start-up and initializes the AccessLib library and all required third party software. If this function returns an error no other AccessLib API calls can be made.

Arguments:

None

Return Value:

- AccessError – The value ACCESS_OK if the library and all third party software are successfully initialized, an error otherwise.

5.3.2 AccessLibShutdown()

The AccessLibShutdown function is called only once before the client application exits to gracefully shutdown the AccessLib library and all third party software.

Arguments:

None

Return Value:

None

5.3.3 getAccessServer()

The getAccessServer function returns an unconnected, unauthenticated instance of the IAccessServer class. The IAccessServer instance returned by this function is owned by the client application and must be explicitly deleted when no longer needed. For the Access Server hosted on the Amazon AWS cloud, the standard host name is “access.agtek.com” and the standard port is 34015.

Arguments:

1. const std::string& host – The name of the machine the Access Server is running on.
2. int port- The TCP/IP port the Access Server is listening on.

Return Value:

- IAccessServer* - An instance of an unconnected, unauthenticated IAccessServer. The pointer is owned by the client application.

5.4 IAccessServer

The IAccessServer class contains all the methods for handling the client's connection to the server. Deleting a connected instance of this class automatically disconnects from the server.

5.4.1 error()

The error method returns the error state of the last communication with the server.

Arguments:

None

Return Value:

- AccessError – The error state of the last communication

5.4.2 errorMessage()

The errorMessage method returns the any error message associated with the last error. Error messages are optional and should not be used to determine if an error occurred. In other words, the lack of an error message does not imply that an error did not occur.

Arguments:

None

Return Value:

- const std::string& - The error message (will be blank if no message)

5.4.3 name()

The name method returns the name and version of the server. This value will only be valid after a successful call to connect.

Arguments:

None

Return Value:

- const std::string& - The server name and version

5.4.4 token()

The token method returns the authenticated users cache-able authentication token. This value will only be valid after the connection has been successfully authenticated.

Arguments:

None

Return Value:

- const std::string& - The users authentication token

5.4.5 connect()

The connect method attempts to create a TCP/IP socket connection to an AGTEK Access Server. If this method returns successfully, the connection will remain open until either it is explicitly closed by the client application with a call to disconnect, or the server determines the connection is idle and closes it automatically. Currently the server timeout is set to approximately one minute, but that value is subject to performance tuning and may change in the future.

Arguments:

None

Return Value:

AccessError – ACCESS_OK, or an error if the connection fails

5.4.6 disconnect()

The disconnect method closes the TCP/IP socket connection to an AGTEK Access Server. Calling disconnect on a connection that has already been closed has no effect.

Arguments:

None

Return Value:

None

5.4.7 authPassword()

The authPassword method authenticates a connection using the user's id and password. This method can only be called after a successful call to connect. If the method succeeds, the server will return a token that can be cached and used instead of the password for future authentication requests. The token can be accessed using the token method.

Arguments:

1. const std::string& id – The user's id
2. const std::string& password – The user's password

Return Value:

- AccessError – ACCESS_OK, or an error if authentication fails

5.4.8 authToken()

The authToken method authenticates a connection using the user's id and cache-able authentication token. This method can only be called after a successful call to connect.

Arguments:

1. const std::string& id – The user's id
2. const std::string& token – The user's authentication token

Return Value:

- AccessError – ACCESS_OK, or an error if authentication fails

5.4.9 setPassword()

The setPassword() method changes the user's password and updates the cache-able authentication token. This method can only be called after a successful call to authPassword or authToken.

Arguments:

1. const std::string& password – The user's new password

Return Value:

- AccessError – ACCESS_OK, or an error if the password change fails

5.4.10 sendEmail()

The sendEmail() method allows a client application to send a simple e-mail (no attachments) through the server. The e-mail will be sent from an account configured on the server. The client application can specify the “to” address, “from” address, subject line and message body. If the “to” or “from” addresses are blank, the server will use a configured default.

Arguments:

1. const std:string& to – The e-mail “to” address, if blank uses server default
2. const std::string& from – The e-mail “from” address, if blank uses server default
3. const std::string& subject – The e-mail subject
4. const std::string& body – The e-mail body

Return Value:

- AccessError – ACCESS_OK, or an error if sending the e-mail fails

5.4.11 getServerTime(time_t & outTime)

This method returns the time the server believes it is. This is useful to compare LicenseKey instance times to make sure the key has not been compromised by changing the time on the client application. This method is also useful as a cheap way to determine if the application has good connectivity with the server.

Arguments:

- outTime – The return value, in Std-C time_t units, of the server time.

Return Value:

- AccessError – ACCESS_OK, or an error if sending the e-mail fails

5.4.12 getAllAnnouncements(AnnouncementVector & outAnnounceList)

This method returns the list of current announcements from the server. Each time the application starts, it must retrieve announcements and display any new announcement to the user. Each announcement object (Announcement) has an **id, product code, expiration time,** and **message.** The message is a simple HTML string (basic formatting) to be displayed to the user. The application should track the last announcement id displayed so it doesn't repeat displaying announcements to the user. The application may also sort out any announcements with a ProductCode not applicable to the application, but must also accept any announcement with the product code "ALL".

Arguments:

- outAnnounceList – The return vector of Announcements current on the server.

Return Value:

- AccessError – ACCESS_OK, or an error if sending the e-mail fails

5.4.13 getFilteredAnnouncements(const int announcementId, const std::string & inProductCode, AnnouncementVector & outAnnounceList)

This method returns only those announcements suitable for the specifics supplied by the application. Returned announcements must be newer than the announcementId, and must applicable to the product code inProductCode. All other aspects of the returned announcements are as described by the method getAllAnnouncements().

Arguments:

- announcementId – The last read announcement from the application. Returned announements will have ids greater than *announcementId.*
- inProductCode – A product code of the current product. This restricts returned announcments to match only this code, or the special code "ALL".
- outAnnounceList – The returned list of announcements.

Return Value:

- AccessError – ACCESS_OK, or an error if sending the e-mail fails

5.4.14 getProductCodes(ProductCodeVector & outProductCodeList)

This method returns a list of known product codes. The application will need this list in order to load LicenseKeys from storage. The application may cache this list in order to read the license when disconnected from the internet, but should update the list whenever a live connection is made since the product code list can change at any time (typically from the introduction of new products).

Arguments:

- outProductCodeList – The returned list of product codes from the server.

Return Value:

- AccessError – ACCESS_OK, or an error if sending the e-mail fails

5.4.15 trackApi()

The trackApi() method returns a pointer to an ITrackApi instance that implements all the GPS tracking functions. The pointer is owned by the IAccessServer instance and should **NOT** be deleted by the client application. This method can only be called after a successful call to connect.

Arguments:

None

Return Value:

- ITrackApi* - A pointer to the GPS tracking API

5.4.16 projectApi()

The projectApi method returns a pointer to a IProjectApi instance that implements all the project management functions. The pointer is owned by the IAccessServer instance and should **NOT** be deleted by the client application. This method can only be called after a successful call to connect and one of the authentication methods.

Arguments:

None

Return Value:

- IProjectApi* - A pointer to the project management API

5.4.17 vehicleApi()

The vehicleApi method returns a pointer to a IVehicleApi instance that implements all the vehicle management functions. The pointer is owned by the IAccessServer instance and should **NOT** be deleted by the client application. This method can only be called after a successful call to connect and one of the authentication methods.

Arguments:

None

Return Value:

- IVehicleApi* - A pointer to the vehicle management API

5.4.18 assnApi()

The assnApi method returns a pointer to a IAssnApi instance that implements all the vehicle-device association management functions. The pointer is owned by the IAccessServer instance and should **NOT** be deleted by the client application. This method can only be called after a successful call to connect and one of the authentication methods.

Arguments:

None

Return Value:

- IAssnApi* - A pointer to the vehicle-device association management API

5.4.19 fileApi()

The fileApi method returns a pointer to a IFileApi instance that implements all the shared file storage functions. The pointer is owned by the IAccessServer instance and should **NOT** be deleted by the client application. This method can only be called after a successful call to connect and one of the authentication methods.

Arguments:

None

Return Value:

- IFileApi* - A pointer to the shared file storage API

5.4.20 licenseApi()

The licenseApi method returns a point to an ILicenseApi instance that implements all the license function. The pointer is owned by the IAccessServer instance and should NOT be deleted by the client application. This method can only be called after a successful call to connect and one of the authentication methods.

Arguments:

None

5.5 ITrackApi

The ITrackApi interface contains all the methods for managing GPS tracks. An instance of this interface is created after successfully connecting to a sever. However, only the upload method can be called before the connection is authenticated. Unauthenticated uploads are allowed for devices that are registered with the server and have been explicitly assigned to a customer.

5.5.1 download()

The download method retrieves GPS positions from the server. The returned data set can be arbitrarily large depending on the time window specified in the call. To prevent the need for the method to allocate memory to handle this, all the GPS positions are returned to the client application using a callback function. This removes the responsibility for allocating storage for the returned data from the download method. The position callback function is defined as:

```
typedef void
(*PositionCallback)(void *clientp, const GpsPos& pos, int index, int total);
```

In addition to the GPS position data, the download function will pass back a client application supplied pointer. The pointer is typed as a "void *" and can be anything the client needs in the callback function.

The amount of data returned is controlled by the method arguments. The serial number specifies the GPS device. The start and end arguments define an inclusive time window for the data. The maximum number of GPS positions returned will never be greater than end – start + 1. The allFields flag controls which fields are returned for each GPS position. Setting the flag to false decreases the number of fields returned and can speed up download time as fewer bytes are sent. When the flag is set to false the server will only send back the time, latitude and longitude fields for each position.

Arguments:

1. const std::string& serial - The serial number of the GPS device
2. int start – The time stamp of the first position
3. int end – The time stamp of the last position
4. bool allFields – Flag controlling which position fields are returned
5. PositionCallback callback – The callback function for each GPS position
6. void* clientp – Client data passed to the callback function

Return Values:

- Access Error – ACCESS_OK, or an error if the download fails

5.5.1.1 Download Track Example

The following example code assumes a connected and authenticated server connection.

```
void callback(void *clientp, const GpsPos& pos, int index, int total) {
  // Callback code for the download operation.
  printf("  %d: %f %f %f (%d of %d)\n", pos.time(),
    pos.latitude(), pos.longitude(), pos.altitude(),
    index+1, total);

  // Client code should store the GPS data before returning
}

void example(IAccessServer* server) {
  // Get the track API from the server
  ITrackApi* tracks = server->trackApi();

  // Specify the GPS device
  std::string serial = "example0";

  // Specify a 10 second time window
  int start = 1262347200; // January 1, 2010 0:00:00 PM GMT
  int end =   1262347210; // January 1, 2010 0:00:10 PM GMT

  // Download the GPS data getting all the fields
  AccessError error = tracks->download(serial, start, end, true, callback, NULL);

  // Check that the download completed successfully
  if (error) {
    printf("Download failed with error %d\n", error);
  }
}
```

5.5.2 upload()

The upload method sends GPS position data to the server. This method can be called on an unauthenticated connection if the serial number argument specifies a GPS device that is registered on the server and explicitly assigned to a customer. If the GPS device has not been registered then the connection must be authenticated so that the data will be associated with the correct customer. In addition to an error code, the method returns the actual number of points successfully uploaded. This value will only be less than the total number of points if a communication error occurs part way through the operation.

Arguments:

1. const std::string& serial – The GPS device serial number
2. int count – The total number of points to upload
3. GpsPos* points – The GPS position data
4. int* uploaded – If not NULL, uploaded will contain the total number of points successfully sent to the server.

Return Values:

- AccessError – ACCESS_OK, or an error if the upload fails

5.5.3 getTracks()

The getTracks method retrieves GPS track data from the server. The tracks can be filtered by serial number, by project handle or both. To retrieve all tracks pass the values ALL_DEVICES and ALL_PROJECTS for the arguments *serial* and *project* respectively. These values are defined in the ITrackApi header file. The client application must also provide a TrackList instance that the retrieved tracks can be appended to.

The server automatically checks each track against the list of associations so that the returned Track instances have the correct asset id. For tracks that do not have an association the asset id will be blank.

Arguments:

1. const std::string& serial – The GPS device serial number, or the value ALL_DEVICES
2. int project – A project handle, or the value ALL_PROJECTS
3. TrackList& tracks – A TrackList instance to hold the retrieved tracks

Return Values:

- AccessError – ACCESS_OK, or an error if the operation fails

5.5.4 moveTrack()

The moveTrack method moves an existing GPS track from one project to another. The track and the destination project must exist on the server. If the *result* argument is not NULL a copy of the the moved track will be returned. The result track pointer is owned by the client application.

Arguments:

1. int track – The track handle
2. int project – The destination project handle
3. Track* result – If not NULL, result will contain a copy of the moved track

Return Values:

- AccessError – ACCESS_OK, or an error if the operation fails

5.5.5 deleteTrack()

The deleteTrack method deletes a GPS track and all associated GPS position data from the server.

Arguments:

1. int handle – The track handle

Return Values:

- AccessError – ACCESS_OK, or an error if the operation fails

5.5.6 getSerialNumbers()

The getSerialNumbers method returns a list of known device serial numbers for the associated customer. These include devices that have been explicitly assigned to the customer and devices that are not assigned but have uploaded GPS or RTK data.

Arguments:

1. SerialList& serials – A SerialList instance to hold the retrieved serial numbers

Return Values:

- AccessError – ACCESS_OK, or an error if the operation fails

5.6 IRtkApi

The IRtkApi interface contains all the methods for managing RTK tracks. An instance of this interface is created after successfully connecting to a sever. However, only the upload method can be called before the connection is authenticated. Unauthenticated uploads are allowed for devices that are registered with the server and have been explicitly assigned to a customer.

5.6.1 download()

The download method retrieves RTK positions from the server. The returned data set can be arbitrarily large depending on the time window specified in the call. To prevent the need for the method to allocate memory to handle this, all the RTK positions are returned to the client application using a callback function. This removes the responsibility for allocating storage for the returned data from the download method. The position callback function is defined as:

```
typedef void
(*RtkPosCallback)(void *clientp, const RtkPos& pos, int index, int total);
```

In addition to the RTK position data, the download function will pass back a client application supplied pointer. The pointer is typed as a "void *" and can be anything the client needs in the callback function.

The amount of data returned is controlled by the method arguments. The serial number specifies the RTK device. The start and end arguments define an inclusive time window for the data. The maximum number of RTK positions returned will never be greater than end – start + 1.

Arguments:

1. const std::string& serial - The serial number of the RTK device
2. int start – The time stamp of the first position
3. int end – The time stamp of the last position
4. RtkPosCallback callback – The callback function for each RTK position
5. void* clientp – Client data passed to the callback function

Return Values:

- Access Error – ACCESS_OK, or an error if the download fails

5.6.1.1 Download Example

The following example code assumes a connected and authenticated server connection.

```
void callback(void *clientp, const RtkPos& pos, int index, int total) {
  // Callback code for the download operation.
  printf("  %d: %f %f %f (%d of %d)\n", pos.time(),
    pos.northing(), pos.easting(), pos.elevation(),
    index+1, total);
```

```
  // Client code should store the GPS data before returning
}

void example(IAccessServer* server) {
  // Get the RTK API from the server
  IRtkApi* rtk = server->rtkApi();

  // Specify the RTK device
  std::string serial = "exampleRTK0";

  // Specify a 10 second time window
  int start = 1262347200; // January 1, 2010 0:00:00 PM GMT
  int end =   1262347210; // January 1, 2010 0:00:10 PM GMT

  // Download the RTK data
  AccessError error = rtk->download(serial, start, end, callback, NULL);

  // Check that the download completed successfully
  if (error) {
    printf("RTK download failed with error %d\n", error);
  }
}
```

5.6.2 upload()

The upload method sends RTK position data to the server. This method can be called on an unauthenticated connection if the serial number argument specifies a RTK device that is registered on the server and explicitly assigned to a customer. If the RTK device has not been registered then the connection must be authenticated so that the data will be associated with the correct customer. In addition to an error code, the method returns the actual number of points successfully uploaded. This value will only be less than the total number of points if a communication error occurs part way through the operation.

Arguments:

5. const std::string& serial – The RTK device serial number
6. int count – The total number of points to upload
7. RtkPos* points – The RTK position data
8. int* uploaded – If not NULL, uploaded will contain the total number of points successfully sent to the server.

Return Values:

- AccessError – ACCESS_OK, or an error if the upload fails

5.6.3 getTracks()

The getTracks method retrieves RTK track data from the server. The tracks can be filtered by serial number, by project handle or both. To retrieve all tracks pass the values ALL_DEVICES and ALL_PROJECTS for the arguments *serial* and *project* respectively. These values are defined in the ITrackApi header file. The client application must also provide a TrackList instance that the retrieved tracks can be appended to.

The server automatically checks each track against the list of associations so that the returned Track instances have the correct asset id. For tracks that do not have an association the asset id will be blank.

Arguments:

4. const std::string& serial – The RTK device serial number, or the value ALL_DEVICES
5. int project – A project handle, or the value ALL_PROJECTS
6. TrackList& tracks – A TrackList instance to hold the retrieved tracks

Return Values:

- AccessError – ACCESS_OK, or an error if the operation fails

5.6.4 moveTrack()

The moveTrack method moves an existing RTK track from one project to another. The track and the destination project must exist on the server. If the *result* argument is not NULL a copy of the the moved track will be returned. The result track pointer is owned by the client application.

Arguments:

4. int track – The track handle
5. int project – The destination project handle
6. Track* result – If not NULL, result will contain a copy of the moved track

Return Values:

- AccessError – ACCESS_OK, or an error if the operation fails

5.6.5 deleteTrack()

The deleteTrack method deletes a RTK track and all associated RTK position data from the server.

Arguments:

2. int handle – The track handle

Return Values:

- AccessError – ACCESS_OK, or an error if the operation fails

5.7 IProjectApi

The IProjectApi interface contains all the methods for managing customer projects. An instance of this interface is created after successfully connecting to a sever and authenticating.

5.7.1 getProjects()

The getProjects() method retrieves the list of customer projects from the server. The *closed* argument is used to determine if closed projects are included in the list.

Arguments:

1. bool closed – Flag to indicate if closed projects should be included in the list
2. ProjectList& projects – A ProjectList instance to hold the returned projects

Return Values:

- AccessError – ACCESS_OK, or an error if the operation fails

5.7.2 addProject()

The addProject method adds a new customer project to the server. An error will be returned if a project with the specified name already exists on the server. If the customer has file storage enabled, this method will also create a folder with the project name in the file storage system in the customer's “Projects” folder. If the *project* argument is not NULL a pointer to the newly created project will be returned. The returned pointer is owned by the client application.

Arguments:

1. const std::string& name – The project name, must be unique for the customer
2. const std::string& description – The project description
3. int startTime – The project start time as a time stamp
4. double neLat – The latitude of the north-east corner of the project
5. double neLon – The longitude of the north-east corner of the project
6. double swLat – The latitude of the south-west corner of the project
7. double swLon – The longitude of the south-west corner of the project
8. Project* project – If not NULL, the pointer will contain the created project

Return Values:

- AccessError – ACCESS_OK, or an error if the operation fails

5.7.3 updateProject()

The updateProject method modifies a project's description, start time and bounding box on the server.

Arguments:

1. Project* project – A pointer to a project with updated fields

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.7.4 deleteProject()

The deleteProject method deletes a project from the server. All tracks associated with the project are moved to the “Other” project. This method does **NOT** delete the file storage folder associated with the project.

Arguments:

1. int handle – The project handle

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.8 IVehicleApi

The IVehicleApi interface contains all the methods for managing vehicles. An instance of this interface is created after successfully connecting to a sever and authenticating.

5.8.1 getVehicles()

The getVehicles method retrieves vehicle information from the server.

Arguments:

1. VehicleList& vehicles - A VehicleList instance to hold the returned vehicles

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.8.2 addVehicle()

The addVehicle method adds a new vehicle to the server. The server will automatically generate a unique asset id for the vehicle. The vehicle fields *status*, *color* and *avgload* were recently added as placeholders for future development. The fields are stored on the server but their meaning has not been clearly defined. Color should be encoded as an integer in the form 0xRRGGBB. If the *vehicle* argument is not NULL a pointer to the newly created vehicle will be returned. The returned pointer is owned by the client application.

Arguments:

1. const std::string& description – The vehicle description
2. int status – The vehicle status (currently unused)
3. int color – The vehicle display color (currently unused)
4. double avgload – The vehicle load in yards (currently unused)
5. Vehicle* vehicle – If not NULL, the pointer will contain the created vehicle

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.8.3 updateVehicle()

The updateVehicle method updates the fields associated with the vehicle asset id.

Arguments:

1. Vehicle* vehicle – A pointer to a vehicle with updated fields

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.8.4 deleteVehicle()

The deleteVehicle method removes the vehicle with the specified asset id from the server.

Arguments:

1. const std::string& assetId – A vehicle asset id

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.9 IAssnApi

The IAssnApi interface contains all the methods for managing vehicle-device associations. An instance of this interface is created after successfully connecting to a sever and authenticating. Associations are used to pair a vehicle with GPS tracks for a defined time window. The server checks the list of associations each time a call to the ITrackApi getTracks method is called. If a matching association is found the returned track will include the vehicle asset id.

5.9.1 getAssns()

The getAssns method retrieves association information from the server.

Arguments:

1. AssnList& assns – An AssnList instance to hold the returned associations

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.9.2 addAssn()

The addAssn method adds a new vehicle-device association to the server. The time window for the association is defined by the *startTime* and *endTime* inclusive time stamps. If the *assn* argument is not NULL a pointer to the newly created association will be returned.

Arguments:

1. const std::string& serial – A device serial number
2. const std::string& assetId – A vehicle asset id
3. int startTime – The starting time stamp for the association
4. int endTime – The ending time stamp for the association
5. Assn* assn – If not NULL, the pointer will contain the created association

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.9.3 updateAssn()

The updateAssn method updates the association fields.

Arguments:

1. Assn* assn – A pointer to an association with updated fields

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.9.4 deleteAssn()

The deleteAssn method removes the vehicle-device association from the server.

Arguments:

1. int handle – The association handle

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10 IFileApi

The IFileApi contains all the methods for accessing the shared file storage system on the server. The shared file storage system models the folder/file hierarchy of a standard file system. Files stored on the server are shared by all of a customer's users. Unlike a standard file system, there is no concept of individual file privileges. Customer files are **NOT** shared between different customer accounts and are **NOT** accessible by AGTEK employees.

Each customer has an unnamed root level folder that contains all the sub-folders and files created by client applications. Since a folder hierarchy can be arbitrarily large client applications can request the entire tree or just the contents of a single folder. There is no limit to the number of files and folders a client application can store. However, each customer account has a maximum storage size that is the maximum number of stored bytes. This value is assigned and controlled by AGTEK support. Additionally, there is a a maximum single file storage size allowed by the storage system. As of the writing of this document that value is approximately 5 gigabytes.

5.10.1 setCaCertFile()

The setCaCertFile method specifies the file containing SSL authority information. The current implementation does not support SSL encrypted TCP/IP sockets. This method currently does nothing.

Arguments:

1. const std::string& file – The full path to a file containing SSL authority information

Return Value:

None

5.10.2 setSecure()

The setSecure method controls the use of SSL encryption over the TCP/IP socket. The current implementation does not support SSL encrypted TCP/IP sockets. This method currently does nothing.

Arguments:

1. bool secure – A flag indicating if SSL should be enabled

Return Value:

None

5.10.3 getAccountInfo()

The getAccountInfo method retrieves account information for the authenticated user from the server. Included in the account information is the total amount of file storage space allocated and the amount that has been used. The returned pointer is owned by the client application and should be deleted when no longer in use.

Arguments:

1. IAccountInfo** info - Pointer to contain the account information

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.4 getRootFolder()

The getRootFolder method retrieves the root folder from the server. If the *recurse* argument is "false" only the files and folders that are the immediate children of the root folder will be returned. Those folders will have their *fetched* flag set to false. The client application will need to make subsequent calls to the getFolder method to retrieve the contents of each child folder. If the *recurse* argument is "true" the server will recurse through all sub-folders returning the entire hierarchy. Depending on the number of files stored this operation can be time consuming. The returned pointer is owned by the client application and should be deleted when no longer in use.

Though the client owns the root folder pointer passed to the method, all child IRemoteFolder and IRemoteFile pointers returned from the IRemoteFolder interface are owned by the root and should **NOT** be deleted by the client application. When the root folder pointer is deleted it will take care of cleaning up memory associated with the hierarchy.

Arguments:

1. bool recurse – A flag indicating whether to recurse through the entire hierarchy
2. IRemoteFolder** root – Pointer to contain the root folder

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.5 getFolder()

The getFolder method retrieves the contents of a folder from the server. The *recurse* argument acts exactly the same as in the method getRootFolder.

Arguments:

1. IRemoteFolder* folder – The folder to fetch the contents of
2. bool recurse - A flag indicating whether to recurse through all the folders children

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.6 createFolder()

The createFolder method creates a new folder as a child of the *folder* argument. If a file or folder with the specified name already exists the error ACCESS_ITEM_EXISTS will be returned. To get a pointer to the newly created folder the client application should call *folder->folder(name)* after this method returns successfully.

Arguments:

1. IRemoteFolder* folder – The parent folder

2. const std::string& name – The name of the new child folder

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.7 deleteFolder()

The deleteFolder method deletes the specified folder and all of it's contents recursively. The *folder* argument will be deleted by the parent IRemoteFolder instance and should not be referenced after this method returns successfully. If the folder has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFolder* folder – The folder to delete

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.8 moveFolder()

The moveFolder method moves a child folder from one parent to another. If the folder has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFolder* folder – The child folder to move
2. IRemoteFolder* dest – The destination parent folder

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.9 renameFolder()

The renameFolder method renames a folder. If the folder has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFolder* folder – The folder to rename
2. const std::string& name – The new folder name

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.10 setFolderDescription()

The setFolderDescription method changes the description field of a folder. If the folder has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be

returned.

Arguments:

1. IRemoteFolder* folder – The folder to set the description of
2. const std::string& description – The folder description

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.11 createFolderLink()

The current server does not support public links to stored folders. This method will always return the error ACCESS_UNIMPLEMENTED.

Arguments:

1. IRemoteFolder* folder – The stored folder

Return Value:

- This method always returns the error ACCESS_UNIMPLEMENTED

5.10.12 deleteFolderLink()

The current server does not support public links to stored folders. This method will always return the error ACCESS_UNIMPLEMENTED.

Arguments:

1. IRemoteFolder* folder – The stored folder

Return Value:

- This method always returns the error ACCESS_UNIMPLEMENTED

5.10.13 uploadFile()

The uploadFile method uploads a local file to the shared file storage system and places it in the specified folder. Uploading a file to the server can take a non-trivial amount of time. In order for the client application to be able to monitor the progress and optionally abort the operation, the method accepts a callback function that will be called periodically. The progress callback returns either the value OP_CONTINUE to continue the operation, or OP_ABORT to abort it. If the operation is aborted the uploadFile method will return the error ACCESS_OP_ABORTED. The callback function includes a pointer to optional client data passed into the uploadFile method. The client data is passed through as a "void *". The callback function is defined in the head file "access/IFileApi.h" as:

```
typedef int (*FileProgressCallback)(void *clientp, double total, double now);
```

The client application can get a pointer to the newly created file after the uploadFile method returns successfully by calling *folder->file(name)*.

Arguments:

1. IRemoteFolder* folder – The parent folder for the file
2. const std::string& name – The name the file will be stored with
3. const std::string& fpath – The full path to the file on the local system
4. FileProgressCallbck cb – A callback for upload progress
5. void* clientp – A generic pointer to client application data

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.13.1 Upload File Example

The following example code assumes a connected and authenticated server connection.

```
int progress(void *clientp, double total, double now) {
  double percent = now * 100.0 / total;
  printf("Progress %d\n", percent);
  return OP_CONTINUE;
}

void upload(IAccessServer* server) {
  // Get the name and path to the file to upload
  std::string name = "SomeFile.txt";
  std::string fpath = "/home/test/upload.txt";

  // Get the IFileApi instance
  IFileApi* files = server->fileApi();

  // Get a pointer to the root folder
  IRemoteFolder* root = NULL;
  AccessError error = files->getRootFolder(false, &root);
  if (!error) {
    error = files->uploadFile(root, name, fpath, progress, NULL);
  }

  if (!error) {
    IRemoteFile* file = root->file(name);
  }

  if (root)
    delete root;
}
```

5.10.14 downloadFile()

The downloadFile method downloads a file from the shared file storage system to the local file system. Downloading a file from the server can take a non-trivial amount of time. In order for the client application to be able to monitor the progress and optionally abort the operation the method accepts a callback function that will be called periodically. The progress callback returns either the value

OP_CONTINUE to continue the operation, or OP_ABORT to abort it. If the operation is aborted the downloadFile method will return the error ACCESS_OP_ABORTED. The callback function includes a pointer to optional client data passed into the downloadFile method. The client data is passed through as a "void *". If the file has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFile* file – The shared file to download
2. const std::string& fpath – The full path to the local file to download to
3. FileProgressCallback cb – A callback for download progress
4. void* clientp – A generic pointer to client application data

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.14.1 Download File Example

The following example code assumes a connected and authenticated server connection.

```
int progress(void *clientp, double total, double now) {
  double percent = now * 100.0 / total;
  printf("Progress %d\n", percent);
  return OP_CONTINUE;
}

void download(IAccessServer* server) {
  // Get the name and path to the file to upload
  std::string name = "SomeFile.txt";
  std::string fpath = "/home/test/download.txt";

  // Get the IFileApi instance
  IFileApi* files = server->fileApi();

  // Get a pointer to the root folder
  IRemoteFolder* root = NULL;
  AccessError error = files->getRootFolder(false, &root);
  if (!error) {
    IRemoteFile* file = root->file(name);
    error = files->downloadFile(file, fpath, progress, NULL);
  }

  if (root)
    delete root;
}
```

5.10.15 deleteFile()

The deleteFile method deletes a file from the shared file storage system. The *file* argument will be deleted by the parent IRemoteFolder instance and should not be referenced after this method returns successfully. If the file has been renamed, moved or deleted by another application the error

ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFile* file – The file to delete from the shared file storage system

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.16 moveFile()

The moveFile method moves a stored file from one parent folder to another. If the file has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFile* file – The stored file to move
2. IRemoteFolder* dest – The destination parent folder

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.17 renameFile()

The renameFile method renames a stored file. If the file has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFile* file – The stored file to rename
2. const std::string& name – The new file name

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.18 setFileDescription()

The setFileDescription method changes the description field of a stored file. If the file has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFile* file – The stored file to set the description on
2. const std::string& description – The new file description

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.19 createFileLink

The createFileLink method creates a public URL that will allow anyone to access the specified stored file using a web browser. After returning successfully, the client application can access the URL by calling *file->link()*. If the file has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFile* file – The stored file to create the public link for

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.10.20 deleteFileLink()

The deleteFileLink method removes a public URL from a stored file. The file will no longer be accessible using a web browser. If the file has been renamed, moved or deleted by another application the error ACCESS_ITEM_NOT_EXIST will be returned.

Arguments:

1. IRemoteFile* file – The stored file to remove the public link from

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.11 ilicenseApi

ADD Examples of

- getting list of keys
- get the list of ProductCodes (may need to document the method)
- checking out a key
- checking in a key
- saving a key
- loading a key

Brief discussion of LicenseKey types; OneTime, Software, Rental, Internet (aka Timed)

The ILicenseApi interface contains all the methods for managing LicenseKey operations. An instance of this interface is created after successfully connecting to a sever and authenticating. LicenseKeys are used to control access to AGTEK programs, dictated by the contract signed between the customer and AGTEK. LicenseKeys are maintained and serviced via the AGTEK Access Server, allowing programs to checkout and checkin keys. Additional functionality for creating keys and user management is available to those users flagged as “customer admin” or “AGTEK support”. This document describes the API methods to accomplish these tasks. It is outside the scope of this document to describe the structure of the AGTEK Access server, how keys are managed internally, or what management tools are available for AGTEK Support use.

When an application interacts with the LicenseApi, there are several objects with which the programmer must be familiar:

- LicenseApi – The API instance for dealing with licenses. Detailed in section 5.11
- AdminApi – The administation API for altering/updating license and user infrormation in the ACCESS Server. Detailed in section 5.12.
- LicenseKey – An object representing a License instance.
- ProductCode – An object representing a simple product code string (e.g. “SD1”) and a descriptive string (e.g. “SmartDirt”).
- LicenseLogEntry – An object representing a log entry for a specific LicenseKey.
- LicenseKeyVector – A stdlib implementation of a vector of LicenseKeys.
- ProductCodeVector – A stdlib implementation of a vector or StorageUsers.
- LicenseLogEntryVector – A stdlib implementation of a vector of LicenseLogEntry.
- StorageUser – An obejct representing a user in the AGTEK Access system.

A note about interface types.

Generally the types used to interface with the Agtek Access API are fairly basic. Most types are restricted to stdlib implementations (e.g. std::string) but in some cases the types are reduced to primitives for convenience (e.g. char *). Generally arrays are not used because of the burden of

memory management. Stdlib vector implementations are used for the collections of objects going in and out.

Example checking a key in and out.

This example assumes that the application has an instance of the AGTEK Access Client in a variable called “server_”. Note that error handling is less than elegant in this example.

```
ILicenseApi * lApi = server_->licenseApi();
std::string PCODE = “DEV1”; // Example product code

if( lApi == NULL )
{
        stdout << "License API instance must not be null";
        return;
}

// List all available licenses, then pick the first timed
// license to work with.
if( lApi->getLicenses( licenseList )  != ACCESS_OK )
  return;

LicenseKey myLicense;

int found_key    = 0;
for( it = licenseList.begin(); it < licenseList.end(); it++ )
{
        LicenseKey lk = *it;

        // Find an available key, which can not be checked out.
        if( ! lk.isCheckedOut() && (lk.hasProductCode(PCODE) )
        {
                found_key = 1;
                myLicense = lk;
                break;
        }
}
if( ! found_timed )
        complainNotEnoughLicenses();

printf( "Found a license %s\n", myLicense.toString().c_str() );

// Now test the functionality.
if( lApi->checkoutLicense(myLicense, PCODE)  != ACCESS_OK )
        return;

// Application does work then decides to renew the key...
if(  lApi->renewLicense(myLicense)  != ACCESS_OK )
        return;

if( lApi->checkinLicense(myLicense) != ACCESS_OK )
        return;
```

5.11.1 getLicenses(LicenseKeyVector & outLicenseList)

This method returns a LicenseKeyVector of all licenses belonging to the current customer and for which the current user is authorized to use. The application will need to search the vector list for licenses matching the product code of the currently running application. The application also needs to select for licenses currently in use, or licenses which are available depending on the use case.

Arguments:

- LicenseKeyVector & outLicenseList – A vector to store the return list.

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.11.2 checkoutLicense(LicenseKey & inLicense, const std::string product)

Checks out the specified LicenseKey, for use in the specified application. Product must be one of the currently support product codes, and the product code must be set on the license.

Arguments:

- LicenseKey & inLicense – The LicenseKey to be checked out.
- const std::string product – The product code of the application using this license.

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.11.3 checkinLicense(LicenseKey & inLicense)

Checks in the specified LicenseKey, for use by other users and/or applications.

Arguments:

- LicenseKey & inLicense – The LicenseKey to be checked in.

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails This method may return the error code ACCESS_LICENSE_CHECKED_IN if the license has already been checked in. In this case the application should consider the license to be released and behave appropriately.

5.11.4 renewLicense(LicenseKey & inLicense)

Renews the checkout period of the LicenseKey. This has the same functionality as if the user checked in the key and immediately check out the key.

Arguments:

- LicenseKey & inLicense – The LicenseKey to be renewed.

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.11.5 submitLogComment(std::string comment, const std::string product)

This method will submit a string comment to the LicenseLog inside the Access Server. This is typically used when no license are available. These strings will be reviewed periodically by AGTEK employees to make sure customers have enough licenses for their use.

Arguments:

- std::string comment – The comment to be submitted.
- std::string product – The product code submitting the comment.

Return Value:

- AccessError – ACCESS_OK, or an error if the operation fails

5.12 LincenseKey Object

The LicenseKey is the primary object used to represent a particular license. The application uses this object to communicate with the server, save license state, and determine the state of the license. A LicenseKey object will be returned from the ILicenseApi methods; getLicenses(), checkoutLicense(), checkinLicense(), renewLicense(). In the case of checkout/checkin/renew, the returned LicenseKey object represents the new state of the LicenseKey.

In general the application will restrict itself to dealing with the “getter” methods on the object. These methods are used to retrieve state information about they LicenseKey. You should be aware that while there are “setter” methods, these do nothing to the “real” key existing on the server, they only modify your local copy and are generally used by the communication protocol in creating the object[1].

5.12.1 getCustomerId()

This method retrieves the server id of the customer. Generally this is not useful for applications, but it used as part of the communication protocol and server operations.

Arguments:

- None.

Return Value:

- The integer value of the customer.

1 The one exception is when updating LicenseKey fields via the management interfaces. In that case the values you may set in are specified by the management function (update key, etc).

5.12.2 setCustomerId(int c)

This method sets the server id of the customer for this LicenseKey. Generally this is not useful for applications, but is used as part of the communication protocol and server operations.

Arguments:

- The server id of the customer.

Return Value:

- void

5.12.3 getId()

This method will return the long (32 bit) value of the LicenseKey's ID.

Arguments:

- None.

Return Value:

- The long value of the LicenseKey's ID.

5.12.4 getIdString()

This method will return the string value of the LicenseKey's ID.

Arguments:

- None.

Return Value:

- The std::string value of the LicenseKey's ID.

5.12.5 getMaxUsers()

This method will return the maximum number of users allowed to be assigned to this key.

Arguments:

- None.

Return Value:

- The int value maximum number of users for this LicenseKey.

5.12.6 getMaxCheckout()

This method returns the maximum amount of time this LicenseKey may be checked out.

Arguments:

- None.

Return Value:

- The time_t value of the LicenseKey's maximum checkout duration.

5.12.7 getStartTime()

This method returns the time this key was created.

Arguments:

- None.

Return Value:

- The time_t value of the LicenseKey's creation time.

5.12.8 getExpireTime()

This method returns the time when this LicenseKey will expire and be no longer useable.

Arguments:

- None.

Return Value:

- The time_t value of the LicenseKey's expiration time.

5.12.9 getProductCodes()

This method returns the list of product codes for which this LicenseKey is applicable.

Arguments:

- None.

Return Value:

- The ProductCodeVector value of the LicenseKey's list of product codes.

5.12.10 getTimedCodes()

This method returns the list of product codes for which this LicenseKey is temporarily applicable. At some point (defined on a per key basis) these product codes will disappear and no longer be applicable.

Arguments:

- None.

Return Value:

- The ProductCodeVector value of the LicenseKey's list of temporary product codes.

5.12.11 getTimedExpire()

This returns the time at which the timed product codes will expire for this LicenseKey.

Arguments:

- None.

Return Value:

- The time_t value of the LicenseKey's timed code expiration time.

5.12.12 isCheckedOut()

This method will return the checked out state of this copy of the LicenseKey. This method will not indicate if the key is checked out on a different computer.

Arguments:

- None.

Return Value:

- TRUE if this copy of the LicenseKey is checked out, FALSE otherwise.

5.12.13 getDueDate()

This method will return the time when the LicenseKey is due, e.g. it's current check out will expire.

Arguments:

- None.

Return Value:

- The time_t value of the time at which the LicenseKey is not longer checked out.

5.12.14 getCheckoutUserId()

This method will return the server id of the user currently checking out the LicenseKey. This method is generally not useful for application use.

Arguments:

- None.

Return Value:

- The int value of the user holding this LicenseKey.

5.12.15 getCheckoutUser()

This method returns the string value of the name of the user currently holding this key.

Arguments:

- None.

Return Value:

- The std::string value of the user holding this LicenseKey.

5.12.16 getCheckoutUserPhone()

This method returns the string value of the phone number of the user currently holding this key.

Arguments:

- None.

Return Value:

- The std::string phone number of the user holding this LicenseKey.

5.12.17 hasProductCode(const std::string& code)

This method will verify if the LicenseKey is usable with the specified product code. Applications make use of this to determine if a LicenseKey is usable for a specified application.

Arguments:

- The std::string value of the application's product code.

Return Value:

- True if the product code is attached to this LicenseKey, False otherwise.

5.12.18 hasProductCode(const char * code)

This method will verify if the LicenseKey is usable with the specified product code. Applications make use of this to determine if a LicenseKey is usable for a specified application.

Arguments:

- The char * value of the application's product code.

Return Value:

- True if the product code is attached to this LicenseKey, False otherwise.

5.12.19 isExpired()

This method indicates if the LicenseKey is expired. If the key is expired, the application must not not use the key for any purpose. It is an error to attempt any key operations (other than certain management operations) upon an expired key.

Arguments:

- None

Return Value:

- True if the LicenseKey has expired, False otherwise.

5.12.20 executionAllowed()

This method takes into account the checkout and expiration time of the LicenseKey to determine if the application may continue execution.

Arguments:

- None

Return Value:

- True if the LicenseKey allows the application to execute, False otherwise.

5.12.21 toString()

This method converts the LicenseKey into a string value, typically used for debugging. This string value is generally not appropriate to show to end users.

Arguments:

- None

Return Value:

- The std::string value of the LicenseKey's value.

5.12.22 compileDifferences(const LicenseKey& other)

This method compares two instances of a LicenseKey and compile the differences into a string. Generally this is used for debugging and is not useful end user presentation.

Arguments:

- None

Return Value:

- The std::string value of the LicenseKey's difference values.

5.12.23 save(const std::string& filename, const std::string& password)

This method will save an instance of a LicenseKey to a file. The entire state of the LicenseKey; checkout state, expiration time, current computer finger print, etc. The saved file is encrypted and must be reloaded using the **load()** method.

Arguments:

- The std::string value of the filename (path) where the file should be stored.
- The std::string value of the password used to encrypted the stored LicenseKey.

Return Value:

- None.

5.12.24 save(const std::string& filename)

This method will save an instance of a LicenseKey to a file. The entire state of the LicenseKey; checkout state, expiration time, current computer finger print, etc. The saved file is encrypted and must be reloaded using the **load()** method. The username of the checked out key will be used as the password.

Arguments:

- The std::string value of the filename (path) where the file should be stored.
- The std::string value of the password used to encrypted the stored LicenseKey.

Return Value:

- None.

5.12.25 save(std::ostream& os, const std::string& password)

This method will save an instance of a LicenseKey to an output stream. The entire state of the LicenseKey; checkout state, expiration time, current computer finger print, etc. The saved stream is encrypted and must be reloaded using the **load()** method. The username of the checked out key will be used as the password. This method allows the application to save the LicenseKey information to a variety of storage locations.

Arguments:

- The std::ostream where the file should be stored.
- The std::string value of the password used to encrypted the stored LicenseKey.

Return Value:

- None.

5.12.26 load(const std::string& fname, const std::string& password, ProductCodeVector& pcCache)

This method will load a previously saved LicenseKey from a file. The application must supply the path, the decryption password, and the list of product codes.The product code list is retrievable from the LicenseApi.

If the load method is successful, the application must then check to make sure that the LicenseKey is still checked out and not expired (you may use **executionAllowed()**). The load may fail do to any number of IO errors, but more importantly if the LicenseKey doesn't match the finger print of the local computer, you should assume that there is no LicenseKey and the application must not run.

Arguments:

- The std::string of the filename (path) where the LicenseKey was stored.
- The std::string of the password used to decrypt the LicenseKey.
- The ProductCodeVector of product codes, needed to read the license file.

Return Value:

- True if the LicenseKey read successfully, False otherwise.

5.12.27 load(std::istream& is, const std::string& password, ProductCodeVector& pcCache)

This method will load a previously saved LicenseKey from an istream. The application must supply the istream, the decryption password, and the list of product codes.The product code list is retrievable from the LicenseApi.

If the load method is successful, the application must then check to make sure that the LicenseKey is still checked out and not expired (you may use **executionAllowed()**). The load may fail do to any number of IO errors, but more importantly if the LicenseKey doesn't match the finger print of the local computer, you should assume that there is no LicenseKey and the application must not run.

Arguments:

- The std::istream where the LicenseKey was stored.
- The std::string of the password used to decrypt the LicenseKey.
- The ProductCodeVector of product codes, needed to read the license file.

Return Value:

- True if the LicenseKey read successfully, False otherwise.

5.12.28 checkinAllowed()

This method indicates if checkin is allowed on this LicenseKey. Some key are not allowed to be checked in; SoftwareKey and OneTime keys. All others may be checked in.

Arguments:

- None

Return Value:

- True if the LicenseKey may be checked in, False otherwise.

5.12.29 renewalAllowed()

This method indicates if the license may be renewed. OneTime license keys are not renewable.

SoftwareKeys must be renewed at least once per year, all other keys are renewable as long as the key has not expired.

Arguments:

- None

Return Value:

- True if the LicenseKey may be renewed, False otherwise.

5.12.30 operator==(LicenseKey& other)

This method compares one instance of a LicenseKey to another. All states of the key must match for one instance to be considered equal to another, including but not limited to; customer, checkout state, LicenseKey id, etc.

Arguments:

- None

Return Value:

- True if the LicenseKey keys are equal, False otherwise.

&

5.12.31 getUsers()

This method returns the list of users assigned to this key. During normal LicenseKey operations this list is empty, but wuill become populated for admin functions. The application is not required to check the user on a key as the server does this when the application asks for a list of keys.

Arguments:

- None

Return Value:

- StorageUserVector of assigned users on the LicenseKey.

5.12.32 setUsers(StorageUserVector& userList)

Set the allowable users on the LicenseKey. The key must then be updated on the server in order for this to apply.

Arguments:

- StorageUserVector of assigned users on the LicenseKey.

Return Value:

- None

5.13 License Management Methods

Most of the LicenseApi manaement methods are only available throught the Java API. However some functionality is required for customer use on desktop programs.

- Change the checkout duration of a LicenseKey
- Add and remove users from the customer account.
- Add and remove users to a LicenseKey instance.
- Maintain user information; password reset, phone number
- View the log events on a LicenseKey

Example – Adding a user

Caveat as with the previous example, error recovery is woefully thin.

```
LicenseKeyVector licenseList;
LicenseKeyVector::iterator licenseIter;

IAdminApi * adminApi = server_->adminApi();
if( adminApi == NULL )
        stdout << "Admin API instance must not be null";

ILicenseApi * lApi = server_->licenseApi();
if( lApi == NULL )
        stdout << "License API instance must not be null";

StorageUserVector userList;
StorageUserVector::iterator userIter;

if( adminApi->getUsers( userList ) != ACCESS_OK )
        return;

// Find a license, and perform some user manipulations.
if( lApi->getLicenses( licenseList ) != ACCESS_OK )
        return;

// We'll screw around with the first one.
LicenseKey lk = licenseList[0];
printf( "User mods to license %s\n", lk.toString().c_str() );

// Remove all current users, except the first one, which is assumed to
// be the current user running this test. (remember, this came from a unit test).
StorageUserVector licenseUsers;
if( adminApi->getLicenseUsers( lk, licenseUsers ) != ACCESS_OK )
        return;

StorageUser myself = userList[0];
for( userIter = licenseUsers.begin(); userIter < licenseUsers.end(); userIter++ )
{
        StorageUser user = *userIter;
```

```
        if( user.getHandle() == myself.getHandle() )
                continue;

        printf( "Removing %s\n", user.toString().c_str() );
        if( adminApi->removeLicenseUser( lk, user ) != ACCESS_OK )
                return;
}

// Add new users
if( adminApi->addUser( "x1@cunit.example.com",
                       "XX1",
                       "Stroustrup",
                       "123-4567",
                       false, // Not CustomerAdmin
                       "pass1",
                       xu )  != ACCESS_OK ) // New user output object
        return;

if( adminApi->addUser( "x2@cunit.example.com",
                       "XX2",
                       "Stroustrup", // Must be related to XX1!
                       "234-4567",
                       false,  // Not CustomerAdmin
                       "pass2",
                       xu )  != ACCESS_OK )  // New user output object
        return;

if( dminApi->addUser( "x33@cunit.example.com",
                       "XX3",
                       "Stroustrup", // It's a whole family of them.
                       "345-4567",
                       false,  // Not CustomerAdmin
                       "pass3",
                       xu )  != ACCESS_OK )  // New user output object
        return;

// Add users to the key
if( adminApi->getUsers( userList ) != ACCESS_OK )
        return;

// Pick some users to be targets
StorageUser u1 = userList[1];
StorageUser u2 = userList[2];
StorageUser u3 = userList[3];

// Add some users
if( dminApi->addLicenseUser( lk, u1 ) != ACCESS_OK ) return;
if( adminApi->addLicenseUser( lk, u2 ) != ACCESS_OK ) return;
if( adminApi->addLicenseUser( lk, u3 ) != ACCESS_OK ) return;

// Users are now persistently added to the key in the server

// Remove same, just for fun.
If( adminApi->getLicenseUsers( lk, licenseUsers )  != ACCESS_OK )
        return;
```

```
if( adminApi->removeLicenseUser( lk, u3 ) != ACCESS_OK )
        return;

// XX3 is no longer on the key
```

Example – Modifying a LicenseKey

```
IAdminApi * adminApi = server_->adminApi();
if( adminApi == NULL )
{
        stdout << "Admin API instance must not be null";
        return;
}

LicenseKeyVector licenseList;
LicenseKeyVector::iterator licenseIter;

ILicenseApi * lApi = server_->licenseApi();
if( lApi == NULL )
{
        stdout << "License API instance must not be null";
        return;
}

if( lApi->getLicenses( licenseList ) != ACCESS_OK )
        return;
LicenseKey license = licenseList[0]; // Muck about with the first one.

time_t oldDuration = license.getMaxCheckout();
time_t newDuration = oldDuration / 2;

if( adminApi->updateLicenseCheckoutDuration( newDuration, license ) != ACCESS_OK )
        return;
```

5.13.1 AccessError addUser(std::string inUserId, std::string inFirstName, std::string inLastName, std::string inPhone, bool inAdmin, std::string inPassword, StorageUser& outUser)

This method allows a user, with Customer Admin[2] privileges, the ability to add a new user for the customer. The application must specify the users id, name, phone, password, and a boolean indicating if the user is to be Customer Admin. Successfully adding a new user returns the user record, or class StorageUser.

2 CustomerAdmin users are able to perform all the management functions listed in this section. Other user types include “Developer” and “Support” among others. These other user types are not available through the C/C++ API, nor are they available to a customer.

Restrictions: Each customer is limited to a maximum number of users and the server will prevent that limit from being exceeded. User id strings must be unique across the AGTEK Access system, and unique across all customers. It is recommended that email ID strings are used, but the server doesn't enforce this. If a UserID is not unique, and error is returned.

Arguments:

- inUserId – The unique string representing the user id of the new user.
- inFirstName – The human readable first name of the user, may be empty.
- inLastName – The human readable last name of the user, may be empty.
- inPhone – The string value of the user's phone number, may be empty.
- inAdmin – True if the user is to be a CustomerAdmin, False for standard users.
- inPassword - The initial password of the user.
- outUser – The returned StorageUser instance of the newly added user.

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

5.13.2 getUsers(StorageUserVector& inOutUserList)

This method returns a list of all existing users for the current customer.

Arguments:

- inOutUser – The returned StorageUser instance of existing users for the customer.

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

5.13.3 deleteUser(StorageUser& inUser)

This method will delete the specified user from the Access server. The customer is also removed from any LicenseKeys to which they have been assigned. File,Folders, and all other data created in AGTEK Access by this user are left untouched.

Arguments:

- inUser – The StorageUser instance of the users to be deleted..

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

5.13.4 updateUser(StorageUser& inUser)

This method will update the normal user fields in the server; first name, last name, phone number, customer admin flag. No other fields will be altered. The currently logged in user must be a CustomerAdmin (or better) to successfully execute this method.

Arguments:

- inUser – The StorageUser instance of the users to be updated..

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

5.13.5 setPassword(StorageUser& inUser, const std::string& inPassword)

This method allows a specific user's password to be updated. The currently logged in user must be a CustomerAdmin (or better) to successfully execute this method.

Arguments:

- inUser – The StorageUser instance of the user to update.

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

5.13.6 getLicenseUsers(LicenseKey& inLicense, StorageUserVector& outUserList)

This method returns the list of users assigned to a specific LicenseKey. Generally this method is only executed for management interfaces when adding and removing users on a LicenseKey instance. The currently logged in user must be a CustomerAdmin (or better) to successfully execute this method.

Arguments:

- inLicense – The LicenseKey instance to interrogate.
- outUserList – The list of users assigned to the specified LicenseKey.

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

5.13.7 removeLicenseUser(LicenseKey& inLicense, StorageUser& inUser)

This method removes the specified user from the LicenseKey. The result of this API is immediate and is persistent in the ACCESS Server. The currently logged in user must be a CustomerAdmin (or better)

to successfully execute this method.

Arguments:

- inLicense – The LicenseKey to update.
- inUser – The StorageUser instance of the user to be removed from the LicensseKey.

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

5.13.8 addLicenseUser(LicenseKey& inLicense, StorageUser& inUser)

This method adds the specified user from the LicenseKey. The result of this API is immediate and is persistent in the ACCESS Server. The currently logged in user must be a CustomerAdmin (or better) to successfully execute this method.

Arguments:

- inLicense – The LicenseKey to update.
- inUser – The StorageUser instance of the user to be added to the LicensseKey.

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

5.13.9 updateLicenseCheckoutDuration(time_t inDuration, LicenseKey& inOutLicense)

This method adjusts the maximum checkout time of the specified LicenseKey. The duration of maximum checkout period will limited by the license expiration. The result of this API is immediate and is persistent in the ACCESS Server. The currently logged in user must be a CustomerAdmin (or better) to successfully execute this method.

Arguments:

- time_t – The new checkout duration.
- inOutLicense – The LicenseKey instance to alter. Upon successful execution of this method, the LicenseKey instance is updated with the new information.

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

5.13.10 getLicenseLog(LicenseKey& inLicense, LicenseLogEntryVector& outLog)

This method returns a vector of log entries for the specified LicenseKey object. Each log entry is time stamped and contains a log entry type, potentially a comment describing what the admin was

attempting to do. These entries may be added to using the submitLogComment method. The currently logged in user must be a CustomerAdmin (or better) to successfully execute this method.

Arguments:

- inLicense – The LicenseKey instance to interrogate.
- outLog – The vector of log entries for the specified LicenseKey.

Return Value:

- AccessError – The error code if the operation was unsuccessfuly, or ACCESS_OK if it was successful.

6 Appendix A: Error Codes

AccessError	Value	Description
ACCESS_OK	0	No error
ACCESS_SERVER_ERROR	1	Internal server error
ACCESS_SERVER_BUSY	2	Server too busy to process request
ACCESS_REQUEST_UNKNOWN	10	Unknown or unimplemented request sent to server
ACCESS_REQUEST_INVALID	11	Invalid request sent to server
ACCESS_MISSING_ARGS	12	The library sent a request missing required arguments
ACCESS_CRC_FAILED	30	CRC check failed on sent or received data
ACCESS_AUTH_FAILED	100	Authentication failed due to invalid user id, password or token
ACCESS_NO_AUTH	101	The connection has not been authenticated
ACCESS_NO_SUCH_PROJECT	200	The specified project does not exist
ACCESS_NO_SUCH_VEHICLE	210	The specified vehicle does not exist
ACCESS_NO_SUCH_ASSN	220	The specified association does not exist
ACCESS_ITEM_EXISTS	300	The specified item already exists
ACCESS_ITEM_NOT_EXIST	301	The specified item does not exist
ACCESS_INSUF_PRIV	310	The authenticated user does not have sufficient privileges for the requested operation
ACCESS_OUT_OF_SPACE	311	Not enough free space for the requested upload operation
ACCESS_FILE_SIZE	320	The requested upload exceeds the maximum single file size
ACCESS_REQUEST_FAILED	1001	The request failed but the server did not provide any error information
ACCESS_RESP_UNEXPECTED	1002	The library received an unexpected response from the server
ACCESS_RESP_INVALID	1003	The library received the wrong response from the server
ACCESS_RESP_PARTIAL	1004	The library received a server response that was missing return values
ACCESS_LICENSE_ERROR	1040	An error occurred with a license operation
ACCESS_NO_LICENSE	1041	No license was returned from the operation
ACCESS_TOO_MANY_LICENSES	1042	Too many licenses returned, expecting only one
ACCESS_LICENSE_CHECKED_IN	1043	The license was already checked in
ACCESS_OP_ABORTED	1100	The operation was aborted from the progress callback
ACCESS_FILE_OPEN	1200	The library was unable to open the local file
ACCESS_FILE_WRITE	1201	The library was unable to write to the local file
ACCESS_FILE_READ	1202	The library was unable to read from the local file
ACCESS_FILE_CLOSE	1203	The library failed to close the local file
ACCESS_UNIMPLEMENTED	2000	The client sent a request that the server that has not implemented
ACCESS_OUT_OF_MEMORY	2001	The library failed to allocate memory
ACCESS_INIT_FAILED	2002	The library failed to initialize
ACCESS_SOCKET_FAILED	2003	The library was unable to initialize a TCP/IP socket

AccessError	Value	Description
ACCESS_HOST_UNKNOWN	2004	The library was unable to resolve the specified host name
ACCESS_CONNECT_FAILED	2005	The library was unable to connect to the server on the specified port
ACCESS_COMM_ERROR	2006	A network error occurred while communicating with the server
ACCESS_RECV_ERROR	2010	A network error occurred while writing to the TCP/IP socket
ACCESS_SEND_ERROR	2011	A network error occurred whiel reading from the TCP/IP socket
ACCESS_SOCKET_CLOSED	2012	The socket was previously closed
ACCESS_UNREADABLE_REQ	2013	The library sent an unreadable request to the server
ACCESS_UNREADABLE_RESP	2014	The library received an unreadable response from the server
ACCESS_UNEXPECTED_RESP	2015	The library received an unexpected response from the server
ACCESS_SSL_UNAVAIL	2100	SSL is not supported on the client platform
ACCESS_SSL_CA_CERTS	2101	The SSL certificate file is missing
ACCESS_SSL_CERT	2102	SSL negotiation failed due to an invalid server certificate