WireAPI
A Standard API for AMQP

version 1.2c3

iMatix Corporation

Revised: 2007/08/24

Contents

1 Cover **1**
1.1 State of this Document . . . 1
1.2 Copyright and License . . . 1
1.3 Abstract . . . 1

2 Introduction **2**
2.1 How The AMQ Protocol Works . . . 2
2.2 Goals and Principles . . . 2
2.3 Technicalities . . . 2
2.4 Basic Architecture . . . 3
2.5 The iCL Class Syntax . . . 4

3 The Connection Class **5**
3.1 amq_client_connection_new . . . 5
3.2 amq_client_connection_destroy . . . 6
3.3 amq_client_connection_auth_plain . . . 6
3.4 Connection Properties . . . 6

4 The Session Class **7**
4.1 amq_client_session_new () . . . 7
4.2 amq_client_session_destroy () . . . 7
4.3 amq_client_session_wait () . . . 7
4.4 Content Access Methods . . . 8
4.5 Protocol methods . . . 8
4.6 Session Properties . . . 8

5 The Content Classes **10**
5.1 Content Properties . . . 10
5.2 Basic Content Properties . . . 10
5.3 Content Body Data . . . 11
5.4 Advanced Content Manipulation . . . 11

6 ASL Field Table Classes **12**
6.1 The Easy Way to Make a Field Table . . . 12
6.2 Field Tables in More Detail . . . 12

7 Error Handling **14**

8 Miscellaneous Topics **15**
8.1 Object Ownership . . . 15
8.2 Synchronous vs. Asynchronous Methods . . . 15

8.3 Single-Threaded Background Processing 15
8.4 Unimplemented Functions 16

Chapter 1

Cover

1.1 State of this Document

This document is an end-user reference aimed at software developers.

1.2 Copyright and License

1.3 Abstract

We describe a standard API for application developers that wish to use the AMQP for messaging services in their applications. WireAPI is a standard cross-language API that gives developers full access to the full functionality of the AMQP protocol.

Chapter 2

Introduction

2.1 How The AMQ Protocol Works

The reader should refer to the AMQ Protocol specifications for a detailed discussion of the AMQ Protocol and architecture. See http://www.amqp.org.

2.2 Goals and Principles

The WireAPI is built using the same technology as the OpenAMQ server, and provides a component-based interface to the AMQ protocol from the client side. It is designed to:

- Support multiple channels over a single connection.
- Provide full asynchronous background operations.
- Work both in single-threaded and multi-threaded applications.
- Provide full access to all AMQP methods except those used for connection and session start-up and shut-down.
- Provide full access to all AMQP method properties.
- Act as the basis for other language APIs based on the WireAPI model.
- Be 100% portable to common platforms.
- Work with any AMQP server implementation.

2.3 Technicalities

It is mainly built using these technologies:

- iMatix iCL for all class definitions.
- iMatix SMT for the protocol agent (see later).
- iMatix ASL for the overall production.
- Apache APR for runtime functions.
- iMatix Boom for building and packaging.

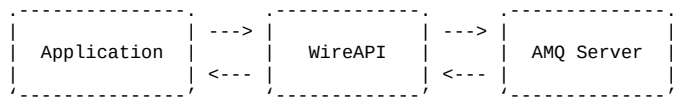
However, the calling application does not need to use any of these tools in order to use WireAPI, except to correctly initialise and terminate iCL. It needs only the header files and the libraries. The simplest way to

get these is to build the full OpenAMQ kernel, and then use the $IBASE/include and $IBASE/lib directories when compiling and linking the application.

The OpenAMQ kernel can be built as a single-threaded or multi-threaded package. The difference is mainly invisible but you may find the single-threaded build easier to debug.

2.4 Basic Architecture

WireAPI sits between the application:



Here is a trivial example that sends one message to an AMQ server. This program does no error handling so you should *not* copy this code:

```
#include "asl.h"
#include "amq_client_connection.h"
#include "amq_client_session.h"
int
main (int argc, char *argv [])
{
    amq_client_connection_t
        *connection = NULL;             //  Current connection
    amq_client_session_t
        *session = NULL;                //  Current session
    amq_content_basic_t
        *content = NULL;                //  Message content
    icl_longstr_t
        *auth_data;
    //  Initialise iCL system
    icl_system_initialise (argc, argv);
    //  Open session to local server
    auth_data  = amq_client_connection_auth_plain ("guest", "guest");
    connection = amq_client_connection_new (
        "localhost", "/", auth_data, "test", 0, 30000);
    icl_longstr_destroy (&auth_data);
    session = amq_client_session_new (connection);
    //  Create a content and send it to the "queue" exchange
    content = amq_content_basic_new ();
    amq_content_basic_set_body       (content, "0123456789", 10, NULL);
    amq_content_basic_set_message_id (content, "ID001");
    amq_client_session_basic_publish (
        session, content, 0, "to-address", NULL, FALSE, FALSE);
    amq_content_basic_destroy (&content);
    //  Shutdown connection and session
    amq_client_session_destroy    (&session);
    amq_client_connection_destroy (&connection);
    //  Terminate iCL system
    icl_system_terminate ();
    return (0);
}
```

This shows some of the main object classes that compose WireAPI:

- amq_client_connection: lets you connect to a server.
- amq_client_session: lets you create a session and talk to the server using this session.
- amq_content_basic: lets you create Basic content to send to the server, or process Basic content received from the server.

2.5 The iCL Class Syntax

The WireAPI classes are all constructed using iCL and some knowledge of iCL will make your life easier. However, the syntax for using iCL classes in C is consistent and we can summarise it:

- classname_new creates a new object instance and returns a reference to that object.
- classname_destroy takes the address of an object reference and destroys that object, if the reference is not null.
- classname_somemethod takes an object reference along with arguments and does some work on the object.
- classname->propertyname accesses a class property.

iCL signals errors by returning a null object (after a new) or by returning a non-zero error code (after other methods).

Thus the correct way to open a connection and session is actually like this:

```
auth_data  = amq_client_connection_auth_plain ("guest", "guest");
connection = amq_client_connection_new (
    "localhost", "/", auth_data, "test", 0, 30000);
icl_longstr_destroy (&auth_data);
if (!connection) {
    icl_console_print ("E: could not open connection");
    return (1);
}
session = amq_client_session_new (connection);
if (!session) {
    icl_console_print ("E: could not open session");
    return (1);
}
```

And the correct way to call a method like publish is:

```
int rc;
...
content = amq_content_basic_new ();
...
rc = amq_client_session_basic_publish (
    session, content, 0, "queue", "mydest", FALSE, FALSE);
amq_content_basic_destroy (&content);
if (rc) {
    icl_console_print ("E: could not publish message");
    return (1);
}
```

Chapter 3

The Connection Class

AMQP is a multi-channel protocol, meaning that one network connection can carry an arbitrary number of parallel, independent virtual connections, which AMQP calls "channels". In WireAPI these are called "sessions" for compatability with other middleware APIs.

Before calling any iCL method including amq_client_connection_new, you must have called icl_system_initialise (), or your application will fail with an abort.

3.1 amq_client_connection_new

Creates a new connection to the server.

```
amq_client_connection_t
    *connection = NULL;             //  Current connection
icl_longstr_t
    *auth_data;                     //  Authentication data
auth_data  = amq_client_connection_auth_plain ("guest", "guest");
connection = amq_client_connection_new (
    "localhost", "/", auth_data, "test", 0, 30000);
icl_longstr_destroy (&auth_data);
if (connection)
    icl_console_print ("I: connected to %s/%s - %s - %s",
        connection->server_product,
        connection->server_version,
        connection->server_platform,
        connection->server_information);
else {
    icl_console_print ("E: could not connect to server");
    return (1);
}
```

- The host_name argument specifies a server name or IP address, optionally ending in ':' plus a port number.
- The virtual_host name specifies the virtual host to which the connection will connect. The default virtual host is "/".
- The auth_data provides an authentication block, used to login to the server. To create a plain-text authentication block, use the auth_plain method. The new method destroys the auth_data block on behalf of the caller.
- The instance argument sets the client instance name, which can be used to identify a specific client in the management console or server log.
- The trace argument sets the trace level for WireAPI.
- The timeout argument governs all synchronous exchanges with the server - if the server does not respond within this time, the connection treats it as a fatal error. A timeout of zero means "infinite".

A good value for fast networks is five to ten seconds; for a slower network, a value of 30 seconds or more is reasonable.

3.2 amq_client_connection_destroy

Closes an open connection, doing a clean shut-down. Applications should use this to end a connection (rather than just exiting):

```
amq_client_connection_destroy (&connection);
```

3.3 amq_client_connection_auth_plain

Returns an authentication block for a plain login:

```
icl_longstr_t
    *auth_data;                     //  Authentication data
auth_data = amq_client_connection_auth_plain ("guest", "guest");
```

3.4 Connection Properties

These are the properties of a connection object:

- alive (Boolean) - FALSE when connection has had an error
- silent (Boolean) - set this TRUE to suppress error reporting
- error_text (string) - error string reported by the API
- reply_text (string) - error string reported by server
- reply_code (integer) - error value reported by server
- version_major (integer) - server protocol version major
- version_major (integer) - server protocol version minor
- server_product (string) - product name reported by server
- server_version (string) - product version reported by server
- server_platform (string) - operatintg system platform reported by server
- server_copyright (string) - copyright notice reported by server
- server_information (string) - other information reported by server

Chapter 4

The Session Class

A session corresponds to an AMQP channel, and is a virtual connection to an AMQ server. You must at least create one session in order to talk with an AMQ server. Multithreaded applications can create many sessions per channel - the advantage is that creating and ending sessions is very fast, much faster than creating a new connection, especially across slow networks and/or firewalls.

4.1 amq_client_session_new ()

Creates a new session:

```
amq_client_session_t
    *session = NULL;                //  Current session
session = amq_client_session_new (connection);
if (!session) {
    icl_console_print ("E: could not open session to server");
    return (1);
}
```

4.2 amq_client_session_destroy ()

Closes an open session, doing a clean shut-down. Applications should call this method when closing a session. Closing the connection is a valid way of closing all open sessions for that connection:

```
amq_client_session_destroy (&session);
```

4.3 amq_client_session_wait ()

Waits for content to arrive from the server. You must call this method in order to get content. Returns zero if content arrived, or -1 if the timeout expired:

```
if (amq_client_session_wait (session, timeout))
    if (session->alive)
        //  timeout expired
    else
        //  session died
else
    //  zero or more contents arrived
```

4.4 Content Access Methods

For each content class (Basic, File, Stream), WireAPI provides a set of methods to access arrived and returned content:

```
amq_client_session_[classname]_[arrived|returned] ()
amq_client_session_[classname]_[arrived|returned]_count ()
```

The first method returns the oldest content waiting to be processed, the second methods returns the number of contents waiting. For example:

```
amq_content_basic_t
    *content;
amq_client_session_wait (session);
if (amq_client_session_basic_arrived_count (session)) {
    icl_console_print ("I: have messages to process...");
    content = amq_client_session_basic_arrived (session);
    while (content) {
        //  process content
        content = amq_client_session_basic_arrived (session);
    }
}
```

When processing arrived or returned content the application must not assume that a single content arrived. It should assume that zero or more contents arrived or returned, and process each of them, and wait again if it needs to.

4.5 Protocol methods

The following methods are directly mapped to the AMQP protocol methods and you should read the AMQP specifications for details:

- amq_client_session_channel_flow - mapped to Channel.Flow.
- amq_client_session_access_request - mapped to Access.Request.
- amq_client_session_exchange_declare - mapped to Exchange.Declare.
- amq_client_session_exchange_delete - mapped to Exchange.Delete.
- amq_client_session_queue_declare - mapped to Queue.Declare.
- amq_client_session_queue_bind - mapped to Queue.Bind.
- amq_client_session_queue_unbind - mapped to Queue.Unbind.
- amq_client_session_queue_purge - mapped to Queue.Purge.
- amq_client_session_queue_delete - mapped to Queue.Delete.
- amq_client_session_basic_consume - mapped to Basic.Consume.
- amq_client_session_basic_cancel - mapped to Basic.Cancel.
- amq_client_session_basic_publish - mapped to Basic.Publish.
- amq_client_session_basic_ack - mapped to Basic.Ack.
- amq_client_session_basic_reject - mapped to Basic.Reject.
- amq_client_session_basic_get - mapped to Basic.Get.

4.6 Session Properties

These are the properties of a session object:

- alive (Boolean) - FALSE when connection has had an error

- silent (Boolean) - set this TRUE to suppress error reporting
- error_text (string) - error string reported by the API
- ticket (integer) - access ticket granted by server
- queue (string) - queue name assigned by server
- exchange (string) - exchange name from last method
- message_count (integer) - number of messages in queue
- consumer_count (integer) - number of consumers
- active (Boolean) - session is paused or active
- reply_text (string) - error string reported by server
- reply_code (integer) - error value reported by server
- consumer_tag (integer) - server-assigned consumer tag
- routing_key (string) - original message routing key
- scope (string) - queue name scope
- delivery_tag (integer) - server-assigned delivery tag
- redelivered (Boolean) - message is being redelivered

Note that all of these except alive, silent, and error_text are the result of methods sent from the server to the client. For detailed descriptions of these properties, read the AMQP specifications. All incoming method arguments are stored as session properties. Thus the "message-count" argument of an incoming Basic.Browse-Ok method will be stored in the message_count property.

Chapter 5

The Content Classes

AMQP uses the term "content" to mean an application message (the term "message" means different things at the application, protocol, and internal technical levels, so is confusing). The protocol defines distinct content classes - basic, file, stream - for different application domains.

For each content class defined in the protocol WireAPI provides an iCL class that lets you create and manipulate contents. For example:

```
amq_content_basic_t
    *content;
content = amq_content_basic_new ();
amq_content_basic_set_body       (content, "0123456789", 10, NULL);
amq_content_basic_set_message_id (content, "ID001");
amq_content_basic_destroy (&content);
```

To create a new content, use the 'new' method. To destroy a content, use the 'destroy' method.

5.1 Content Properties

All contents have these properties, which you can inspect directly:

- class_id (integer) - the content class ID.
- body_size (integer) - the body size of the content.
- exchange (string) - the exchange to which the content was published.
- routing_key (string) - the original routing_key specified by the publisher.

Additionally, each content class defines a set of properties which are noted in the AMQP protocol specifications.

5.2 Basic Content Properties

Basic content has these specific properties:

- content_type (string) - MIME content type.
- content_encoding (string) - MIME content encoding.
- headers (field table) - message header field table.
- delivery_mode (integer) - non-persistent or persistent.
- priority (integer) - message priority, 0 to 9.

- correlation_id (string) - application correlation identifier
- reply_to (string) - the destination to reply to.
- expiration(string) - expiration specification.
- message_id (string) - the application message identifier.
- timestamp (integer) - message timestamp.
- type (string) - message type name.
- user_id (string) - creating user id.
- app_id (string) - creating application id.

To set any of a basic content's properties, do not modify the property directly but use the method:

```
amq_content_basic_set_[propertyname] (content, newvalue)
```

5.3 Content Body Data

To set a content's body, use this method (depending on the content class):

```
amq_content_[class]_set_body (content, byte *data, size_t size, free_fn)
```

Where the free_fn is a function of type 'icl_mem_free_fn *' (compatible with the standard library free() function). If free_fn is not null, it is called when the data needs to be destroyed (when the content is destroyed, or if you call _set_body() again.

To get a content's body, use this method:

```
amq_content_[class]_get_body (content, byte *buffer, size_t limit)
```

Where the buffer is at least as large as content->body_size. This method returns the number of bytes copied, or -1 if the buffer was too small.

5.4 Advanced Content Manipulation

To work with large contents - which do not fit into memory - you must use a more complex API to read and write contents. For details of this please read the content class (.icl) and look at the test case, which demonstrates how to read and write content bodies in frames rather than as single buffers.

Chapter 6

ASL Field Table Classes

6.1 The Easy Way to Make a Field Table

WireAPI uses ASL field tables as a basic type to implement all "field table" arguments. There are two classes that you can use:

- asl_field - defines a single named field holding data of various types.
- asl_field_list - defines a field table (implemented as a list).

The simplest way to build a field table is to construct this using the asl_field_list_build() method:

```
icl_longstr_t
    field_table;
field_table = asl_field_list_build (
    "host",  "Sleeper Service",
    "guest", "My Homework Ate My Dog",
    NULL);
...
icl_longstr_destroy (&field_table);
```

The build() method has a limitation - it only handles string fields. For most AMQP applications this is fine but we can get the same result using calls to individual methods:

```
asl_field_list_t
    *field_list;
icl_longstr_t
    *field_table;
field_list = asl_field_list_new (NULL);
asl_field_new_string  (field_list, "host", "Sleeper Service");
asl_field_new_string  (field_list, "guest", "My Homework Ate My Dog");
field_table = asl_field_list_flatten (field_list);
asl_field_list_destroy (&field_list);
...
icl_longstr_destroy (&field_table);
```

6.2 Field Tables in More Detail

Field tables are held in two ways:

1. As a list of fields. You can navigate this list using standard iCL list navigation commands (first, next, pop, etc.)
2. As a serialised block of data, held in an icl_longstr_t. Field tables held in this format are portable, and can be sent to other machines. This is the format we send across a network via AMQP.

To convert from a field table string to a list, create a new list and pass the string as an argument. To convert from a field list to a table string, use the flatten() method as shown above.

The main asl_field_list methods are:

- asl_field_list_t *asl_field_list_new (icl_longstr_t *string) - create a new field table given a serialised string, or NULL to create an empty field table.
- asl_field_t *asl_field_list_search (list, fieldname) - look for a field with a given name. Note that you must use the unlink() method on the returned field reference when you are finished using it.
- asl_field_list_print (list) - print the field table contents for debugging purposes.
- icl_longstr_t *asl_field_list_build (...) - build a field table from a list of name/value pairs, ending in a null name.

The main asl_field methods are:

- asl_field_new_string (list, fieldname, stringvalue) - create a new string field with the given name and value.
- asl_field_new_integer (list, fieldname, integervalue) - create a new integer field with the given name and value.
- asl_field_new_decimal (list, fieldname, integervalue, decimals) - create a new decimal field with the given name and value.
- asl_field_new_time (list, fieldname, timevalue) - create a new time field with the given name and value.
- asl_field_list_destroy (&list) - destroy a field table.
- asl_field_string (field) - return a string value for a field, doing any necessary conversion.
- asl_field_integer (field) - return an integer value for a field, doing any necessary conversion.

For more details on these methods, refer to the class documentation and/or documented class source code.

You can also access a field's properties directly:

- name (string) - the field name.
- type (character) - 'S', 'I', 'D', or 'T' for string, integer, date, or time field.
- decimals (integer) - number of decimals for a decimal field.
- string (icl_longstr_t *) - string value for the field.
- integer (int64_t) - integer value for the field.

Chapter 7

Error Handling

WireAPI returns two types of error:

- Errors reported internally, e.g. timeout.
- Errors reported by the remote server, e.g. invalid queue name.

In the first case, the error message is provided in session->error_text. There is no localisation - error messages are English only (for now). When there is an internal error, the session->reply_code will be zero.

In the second case, the error message is provided in session->reply_text and an error code is provided in session->reply_code. If the error was at the connection level, the error is placed in connection->reply_text and connection->reply_code instead.

The application can print the right error message using code like this:

```
if (s_session->reply_code)
    icl_console_print ("E: %d - %s",
        s_session->reply_code, s_session->reply_text);
else
    icl_console_print ("E: %s", s_session->error_text);
```

Chapter 8

Miscellaneous Topics

8.1 Object Ownership

WireAPI uses the standard iCL model to define object ownership:

- The layer which calls the "new" method on an object also destroys it.
- If an intermediate layer wants to co-own the object, it does this using possession. An object must explicitly allow this. The content objects are designed for this.
- If an intermediate layer wants to hold a reference to an object, it does this using linking. An object must explicitly allow this.

8.2 Synchronous vs. Asynchronous Methods

AMQP (like all ASL protocols) divides methods strictly into those that expect an immediate reply (synchronous) and those that do not (asynchronous). WireAPI handles these cases as follows:

- When you use a synchronous method (e.g. Basic.Consume), WireAPI waits for the server to respond with a synchronous reply, and it processes this reply. Any asynchronous methods that the server sends before the reply are also processed (e.g. incoming content will be correctly queued.)
- When you use an asynchronous method (e.g. Basic.Publish), WireAPI does not wait for any server response. You can therefore send such events rapidly and expect WireAPI to return as fast as it can, and process them in the background.

8.3 Single-Threaded Background Processing

WireAPI works in both a multi-threaded model (in which one thread handles all dialogue with the server and a second handles the application) and single-threaded (in which a single thread does all the work).

You choose the model when you build OpenAMQ, which specifically has both single- and multi-threaded capability built into it.

The single-threaded model has one specific requirement: the application must periodically call the amq_client_session_wait() method, since it is during this call that asynchronous incoming methods are processed. Always call this method instead of "sleep" or an equivalent in your application.

8.4 Unimplemented Functions

The current OpenAMQ kernel implementation does not support:

- Multilevel contents (which AMQP allows).
- Large parts of the AMQP protocol, including: the File, Stream, Tx, Dtx classes, and acknowledgements.
- Disk-based contents - we assume all contents fit into memory.

These will be added progressively.