Developer's Guide to ASL
The Abstract Syntax Layer

version 1.0

Pieter Hintjens <ph@imatix.com>

Revised: 2007/01/31

Contents

1 Cover

1.1 Copyright and License

1.2 Abstract

ASL is a toolkit for constructing protocol clients and servers. This document explains how to use ASL to build client and server stacks in a variety of programming languages.

1.3 Authors

This document was written by Pieter Hintjens.

2 Introduction

2.1 Aim of this document

ASL is a toolkit for constructing protocol clients and servers. This document explains how to use ASL to build client and server stacks in a variety of programming languages.

The reader should be familiar with underlying iMatix technology, most importantly XNF, Boom, iCL, iPR, and SMT.

2.2 Overview of ASL

2.2.1 What is ASL?

The Abstract Syntax Layer (ASL) is a framework for specifying and building protocol stacks. The ASL framework is based on a protocol specification grammar (asl), tools that validate and interpret this grammar, and a set of target code generators that produce outputs from protocol specifications written in the ASL grammar.

The current set of targets are broadly aimed at multi-channel connected protocols. The AMQP protocol (for which ASL was developed) is a specific instance of such a protocol. Targets cover functionality ranging from basic serialisation code to full frameworks for client APIs and server implementations.

ASL is designed to be easy to extend and modify; the range of code generation targets and their functionality is not pre-defined. The only fixed part of the whole ASL framework is the ASL grammar itself.

Note that the ASL grammar is not fully compatible with the AMQP protocol, which was forked and modified separately starting from June 2006. The first public release of AMQP had no concept of grammar at all.

2.2.2 What is an ASL specification?

In gross terms, an ASL specification consists of a set of class and method definitions, plus a target. The classes and methods provide a structure for the protocol. The target specifies the name of a back-end code generator that will turn the specifications into usable source code.

The ASL framework is roughly a front-end that parses, denormalises, and checks the protocol specifications, plus a set of back-ends that turn these specifications into usable source code.

This front-end/back-end architecture is based on the XNF model oriented programming (MOP) concept. ASL is built using MOP and anyone wanting to modify or extend ASL should have prior MOP experience.

2.2.3 ASL targets

At code generation time, the protocol developer specifies a "target". The target defines entirely what kind of code generation the framework will do. ASL itself defines no fixed architecture on the produced code, though we do provide a standard transport layer and wire-level framing which is explained at the end of this document.

These targets have been developed:

1. A standard C stack (stdc), which produces a full client stack (also called an API), and a server stack into which the developer adds custom code.
2. Prototype stacks in perl, Java, and C#. These generate only the client stacks.

3. A documentation target (doc) that produces extensive documentation for the protocol.
4. A PAL target (pal) that produces a scripting language for the protocol. PAL is documented separately.

Each target is named 'asl_xxx.gsl' where 'xxx' is the target name. So, you can see how the stdc target works by reading asl_stdc.gsl.

2.2.4 Naming conventions

The only naming conventions imposed by ASL are the extensions for files. File names themselves can be defined by the protocol developer. Note that complex protocols need large numbers of files, and the names used can be quite important for clarity.

2.2.5 ASL base specifications

The ASL framework includes a set of "base specifications", which are an optional foundation for multi-channel connected protocols. The base specifications (asl_base.asl) define:

- A set of constants, including error codes (asl_constants.asl).
- The connection class and methods (asl_connection.asl).
- The channel class and methods (asl_channel.asl).
- A number of standard domains such as reply-code.

The base specifications are not mandatory, but they are intended to make the development of certain classes of protocol faster and easier. The base specifications use these files:

- asl_base.asl - base specifications.
- asl_constants.asl - protocol constants.
- asl_connection.asl - the 'connection' class.
- asl_channel.asl - the 'channel' class.

The connection and channel classes together provide a level of functionality that can be compared to protocol frameworks like BEEP (IETF RFCs 3080, 3081).

The connection class provides methods for a client to establish a network connection to a server, and for both peers to operate the connection thereafter.

The channel class provides methods for a client to establish a virtual connection - a channel - to a server and for both peers to operate the virtual connection thereafter.

2.2.6 The chassis concept

ASL is used explicitly for asymmetric client-server protocols, and uses the concept of 'chassis' to encapsulate these roles. In a protocol specification, functionality can be tied to either or both chassis, telling the code generators, "output this code in the appropriate form for the current chassis being generated".

For example, if a particular method is sent by the client to the server but not vice-versa, sending code would be generated in the client stack, and validating code in server stack. The chassis information is not used by all targets.

2.2.7 Specification hierarchy

ASL uses the MOP inheritence technique very heavily. That is, one ASL file can inherit from other files. ASL files can also include other ASL files. Inclusion copies specifications from one file into an other, and

makes it simpler to break large grammars into smaller pieces. Inheritence lets the developer create variations on a protocol.

A typical protocol specification (like the demo protocol examined below) uses both these techniques to create a hierachy of protocol specification files.

We can look at some specific examples from the demo protocol:

- The two chassis specifications inherit from the same base protocol specification called demo.asl. This lets the chassis specifications share the same classes and methods, and layer ASL code on top of those.
- Each class is defined as a seperate .asl file, and included into the base protocol specification. This lets the author edit each class separately; simpler and more convenient than editing the whole protocol as a single file.

3 Worked Example

3.1 The Demo protocol

The ASL project includes a complete worked example, 'demo', which uses the base specifications. The demo protocol implements a simple message routing service, based on the concepts of 'exchange' and 'queue', as developed in AMQP.

We provide a summary of the demo protocol specifications. The demo protocol uses the stdc target, which uses two chassis specifications, one for the server and one for the client.

3.2 Helicopter overview

The specifications themselves are composed of these ASL files:

- demo_client_proto.asl - chassis specification for the client.
- demo_server_proto.asl - chassis specification for the server.
- demo.asl - demo protocol base specification.
- demo_access.asl - 'access' class.
- demo_basic.asl - 'basic' class.
- demo_exchange.asl - 'exchange' class.
- demo_queue.asl - 'queue' class.

The demo.asl file inherits from asl_base.asl, so incorporates the connection and channel classes from that base specification. The client and server chassis inherit from demo.asl, so include all its classes, plus those from asl_base.asl.

The two chassis specifications also inherit from these files, which in the current implementation implement rules for the 'stdc' target:

- asl_client.asl - base specification for a client chassis.
- asl_server.asl - base specification for a server chassis.

To generate code, we issue these commands:

```
gsl demo.asl
gsl demo_client_proto.asl
gsl demo_server_proto.asl
```

In the ASL project.pdl, these three files are defined using the "gsl data" or "gsl public data" classes, which issue the above commands automatically at build time. (The difference between the classes is that the public class also installs the file into the $IBASE/bin directory for reuse by further applications.)

3.3 Protocol specifications

We look at each part of the demo protocol specifications in more detail.

3.3.1 Chassis specification for client

The client chassis specification (demo_client_proto.asl) is the top-level specification for generating the client stack. This specification inherits from demo.asl and asl_client, and then adds the information needed to generate the client stack properly:

```
<?xml version="1.0"?>
<protocol comment = "ASL demo client" script  = "asl_gen" chassis = "client">
<inherit name = "demo" />
<inherit name = "asl_client" />
```

The specification sets two options, the first specifies the name that is used to identify clients on the wire, and the second tells the stdc target to generate a "synchronous client", that is, one which can be used from normal applications. We explain the stdc back-end later.

```
<option name = "product_name" value = "ASL Demo Client" />
<option name = "syncapi" value = "1" />
```

The specification declares defaults for a method (this is to test the use of defaults):

```
<class name = "queue">
<defaults method = "declare">
    <field name = "ticket"       default = "0" />
    <field name = "passive"      default = "0" />
    <field name = "durable"      default = "0" />
    <field name = "exclusive"    default = "0" />
</defaults>
</class>
```

The specification then provides code implementations for methods. This is how the protocol implementor adds specific code to the generated client stack:

```
<class name = "basic">
<action name = "get-ok">
    demo_content_$(class.name)_set_routing_key (
        self->content, method->exchange, method->routing_key, 0);
    demo_content_$(class.name)_list_push_back (
        session->arrived_$(class.name)_list,
        self->content);
</action>
</class>
</protocol>
```

3.3.2 Chassis specification for server

The server chassis specification (demo_server_proto.asl) is the top-level specification for generating the stack. This specification inherits from demo.asl and asl_server, and then adds the information needed to generate the server stack properly:

```
<?xml version="1.0"?>
<protocol
    comment = "ASL demo server"
    script  = "asl_gen"
    chassis = "server"
    >
<inherit name = "demo" />
<inherit name = "asl_server" />
<option name = "product_name" value = "ASL Demo Server" />
```

The specification provides code implementations for methods. This is how the protocol implementor connects the generated server stack with custom-written classes that do the actual work. This interface is part of the stdc target and is explained in detail later:

```
<class name = "channel">
    <action name = "flow">
        channel->active = method->active;
    </action>
</class>
...
</protocol>
```

3.3.3 Demo protocol base specification

The demo protocol base specification (demo.asl) defines the Demo protocol without any chassis or target-specific data. We use the protocol base specification to generate documentation, for example.

The specification inherits from asl_base.asl, the ASL base specification. It includes all the protocol classes (this is just a way of keeping the text readable and editable). Then, it defines a set of options that are used in the generation of the client and server layers:

```
<?xml version="1.0"?>
<protocol
    name     = "demo"
    comment  = "ASL demo protocol"
    script   = "asl_gen"
    target   = "stdc"
    >
<inherit name = "asl_base" />
<include filename = "demo_access.asl" />
<include filename = "demo_exchange.asl" />
<include filename = "demo_queue.asl" />
<include filename = "demo_basic.asl" />
<option name = "protocol_name"     value = "DEMO" />
<option name = "protocol_port"     value = "7654" />
<option name = "protocol_class"    value = "1" />
<option name = "protocol_instance" value = "1" />
<option name = "protocol_major"    value = "1" />
<option name = "protocol_minor"    value = "1" />
```

The file then defines a set of domains that are shared by all classes in this protocol. Domains can be defined in any of the class files as well; it is just convenient to place these in the protocol base specification:

```
<domain name = "access ticket" type = "short">
...
</domain>
...
</protocol>
```

3.3.4 The access class

The demo_access.asl file defines the access class, a set of methods that do access control work. The class file defines the name and index and continues with a mix of documentation, chassis rules, and method specifications. The specific grammar for a class is explained in detail later.

```
<?xml version="1.0"?>
<class
    name    = "access"
    handler = "connection"
    index   = "30"
>
work with access tickets
<doc>
The protocol control access to server resources using access tickets.
A client must explicitly request access tickets before doing work.
An access ticket grants a client the right to use a specific set of
resources - called a "realm" - in specific ways.
</doc>
<doc name = "grammar">
    access              = C:REQUEST S:REQUEST-OK
</doc>
<chassis name = "server" implement = "MUST" />
<chassis name = "client" implement = "MUST" />
<method name = "request" synchronous = "1" index = "10">
request an access ticket
... (method body ommitted for brevity)
</method>
<method name = "request-ok" synchronous = "1" index = "11">
grant access to server resources
... (method body ommitted for brevity)
</method>
</class>
```

3.3.5 The exchange class

The demo_exchange.asl file defines the exchange class, a set of methods to work with server-side exchange entities. In the demo protocol, exchanges are simple routing entities that direct messages into message queues.

```
<?xml version="1.0"?>
<class
    name    = "exchange"
    handler = "channel"
    index   = "40"
>
work with exchanges
<doc>
Exchanges match and distribute messages across queues.  Exchanges can be
configured in the server or created at runtime.
</doc>
<doc name = "grammar">
    exchange            = C:DECLARE S:DECLARE-OK
                        / C:DELETE  S:DELETE-OK
</doc>
<chassis name = "server" implement = "MUST" />
<chassis name = "client" implement = "MUST" />
<method name = "declare" synchronous = "1" index = "10">
declare exchange, create if needed
... (method body ommitted for brevity)
</method>
<method name = "declare-ok" synchronous = "1" index = "11">
confirms an exchange declaration
... (method body ommitted for brevity)
</method>
<method name = "delete" synchronous = "1" index = "20">
delete an exchange
... (method body ommitted for brevity)
</method>
<method name = "delete-ok" synchronous = "1" index = "21">
confirm deletion of an exchange
... (method body ommitted for brevity)
</method>
</class>
```

3.3.6 The queue class

The demo_queue.asl class defines the queue class, a set of methods to work with server-side message queues. In the demo protocol, message queues hold messages until they can be consumed by applications.

```
<?xml version="1.0"?>
<class
    name    = "queue"
    handler = "channel"
    index   = "50"
>
work with queues
<doc>
Queues store and forward messages.  Queues can be configured in the server
or created at runtime.  Queues must be attached to at least one exchange
in order to receive messages from publishers.
</doc>
<doc name = "grammar">
    queue               = C:DECLARE  S:DECLARE-OK
                        / C:BIND     S:BIND-OK
                        / C:CANCEL   S:CANCEL-OK
                        / C:PURGE    S:PURGE-OK
                        / C:DELETE   S:DELETE-OK
</doc>
<chassis name = "server" implement = "MUST" />
<chassis name = "client" implement = "MUST" />
<method name = "declare" synchronous = "1" index = "10">
declare queue, create if needed
... (method body ommitted for brevity)
</method>
<method name = "declare-ok" synchronous = "1" index = "11">
confirms a queue definition
... (method body ommitted for brevity)
</method>
```

```
<method name = "bind" synchronous = "1" index = "20">
bind queue to an exchange
... (method body ommitted for brevity)
</method>
<method name = "bind-ok" synchronous = "1" index = "21">
confirm bind successful
... (method body ommitted for brevity)
</method>
<method name = "purge" synchronous = "1" index = "30">
... (method body ommitted for brevity)
</method>
<method name = "purge-ok" synchronous = "1" index = "31">
confirms a queue purge
... (method body ommitted for brevity)
</method>
<method name = "delete" synchronous = "1" index = "40">
delete a queue
... (method body ommitted for brevity)
</method>
<method name = "delete-ok" synchronous = "1" index = "41">
confirm deletion of a queue
... (method body ommitted for brevity)
</method>
</class>
```

3.3.7 Base specification for client chassis

The asl_client.asl base class defines common specifications for all protocol client layers that are generated from the default backends (which generate ANSI C layers). Thus all protocols (demo included) start with this base set of specifications.

```
<?xml version="1.0"?>
<protocol
    comment = "ASL standard client actions"
    abstract = "1"
    >
```

The specifications are defined per class. For the connection class, we define a set of properties, called the "context", that exist for each client-side connection. Note that these are populated by the pre-supplied client protocol stack implementation.

```
<class name = "connection">
<context>
    icl_shortstr_t
        server_host;
    icl_shortstr_t
        server_name;
    icl_shortstr_t
        server_product;                 //  Reported by server
    icl_shortstr_t
        server_version;
    icl_shortstr_t
        server_platform;
    icl_shortstr_t
        server_copyright;
    icl_shortstr_t
        server_information;
    icl_shortstr_t
        server_instance;
</context>
```

The base specification, like the chassis specification that is derived from it, can specify code to be executed when protocol methods are received. In the following example we specify code to be executed when the server sends us a Connection.Start method. Note that the ASL entity is "action", which can be read as "action to take when this method is received".

```
<action name = "start">
    <local>
    asl_field_list_t
        *fields;                        //  Decoded responses
    </local>
    //
    fields = asl_field_list_new (method->server_properties);
    if (fields) {
```

```
            asl_field_list_cpy (fields, connection->server_host,           "host");
            asl_field_list_cpy (fields, connection->server_instance,       "instance");
            asl_field_list_cpy (fields, connection->server_product,        "product");
            asl_field_list_cpy (fields, connection->server_version,        "version");
            asl_field_list_cpy (fields, connection->server_platform,       "platform");
            asl_field_list_cpy (fields, connection->server_copyright,      "copyright");
            asl_field_list_cpy (fields, connection->server_information,    "information");
            asl_field_list_destroy (&fields);
        }
    </action>
```

All contexts and actions are inherited to the derived chassis class, which can add further context variables, and further class actions.

3.3.8 Base specification for server chassis

The asl_server.asl base class defines common specifications for all protocol server layers that are generated from the default backends (which generate ANSI C layers). Thus all protocols (demo included) start with this base set of specifications.

```
<?xml version="1.0"?>
<protocol
    comment = "ASL standard server actions"
    abstract = "1"
    >
```

The specifications are defined per class. For the server chassis, we define a set of actions; this is a dispatcher that tells the generated code how to implement a given protocol method:

```
<class name = "connection">
<action name = "start-ok">
    $\(basename)_connection_start_ok (connection, method);
</action>
<action name = "tune-ok">
    $\(basename)_connection_tune_ok (connection, method);
</action>
<action name = "open">
    $\(basename)_connection_open (connection, method);
</action>
</class>
```

The $\(basename) symbol is provided by one of the hand-written server classes - the channel class.

If we do not specify an action for a protocol method, the method has no effect. Most often - as in the above example - we pass the protocol method to some hand-written code that implements the necessary work.

3.4 Required hand-written files

The stdc target assumes a number of hand-written files, apart from the ASL model files needed to define the protocol and the chassis. The hand-written files we defined for demo protocol implementation can be used as a basis for other protocols.

3.4.1 Hand-written files for client stack

These hand-written client stack files are assumed by the stdc target and must be provided by the developer:

- demo_client_classes.icl - collection of all classes used in client stack. You can add any hand-written classes into this file to ensure they are properly linked into the client stack library and available to hand-written code in the client chassis (the method actions).
- demo_client_channel.icl - implementation of client channel class. This defines the basename symbol that is needed to generate proper filenames for the client stack.
- demo_client_config.opf - OPF configuration specification for client stack. This file provides the configuration framework for the client stack.

The prefix for these file ('demo') must match the 'protocol_name' option specified in the protocol base specification (the demo.asl file, in this case).

We recommend you copy these files to your own protocol project. These files could in theory be generated, but that would make it hard to insert manual code. Copying and editing the files is the current solution, not the cleanest, but simple.

This file is optional, but useful, and normally copied from ASL at the same time as the above files:

- demo_cli.c - test program for demo protocol.

3.4.2 Hand-written files for server stack

These hand-written server stack files are assumed by the stdc target and must be provided by the developer:

- demo_server_classes.icl - collection of all classes used in server stack.
- demo_server_connection.icl - implementation of server connection class.
- demo_server_channel.icl - implementation of server channel class.
- demo_broker.icl - implementation of server broker class.
- demo_server_config.opf - OPF configuration specification for server stack.

The prefix for these file ('demo') must match the 'protocol_name' option specified in the protocol base specification (the demo.asl file, in this case).

These files are optional but useful, since they define the main server program:

- demo_srv.c - example mainline program.
- demo_server_main.inc - actual mainline code.
- demo_srv_base.cfg - configuration data used by default.

These files are additionally used by the demo protocol server chassis (the classes are included from demo_server_classes, and the methods they contain are referred to in demo_server_proto.asl:

- demo_exchange.icl - exchange class.
- demo_queue.icl - queue master class.
- demo_queue_basic.icl - basic queue class.
- demo_queue_list.icl - queue list class.

3.5 Code generation process

The demo protocol uses the stdc target. This defines two stacks, one for the client and one for the server. Each of these is generated independently. There are also shared files, which are generated from the demo protocol base specification, the demo.asl file.

3.5.1 Generating the client stack

To generate the client stack manually we issue this gsl command:

```
gsl demo_client_proto.asl
```

In a PDL project file, this is expressed as:

```
<file name = "demo_client_proto.asl" class = "gsl data" />
```

The generated client stack consists of these files:

- demo_client_agent.smt - the client-side protocol state machine.
- demo_client_connection.icl - the connection class.
- demo_client_session.icl - the session class.
- demo_client_method.icl - the client method dispatcher class.

We explain the purpose of each of these in more detail when we explain how the stdc target is constructed.

These files are all models, and themselves need to be "compiled" via code generation. When we do this fully, we get this additional set of generated files for the client stack (with indentation showing the relationship between further generated files and their parents):

```
demo_client_agent.smt
    demo_client_agent.c
    demo_client_agent.h
demo_client_channel.icl
    demo_client_channel.c
    demo_client_channel.h
    demo_client_channel_test.c
    demo_client_channel_table.icl
        demo_client_channel_table.c
        demo_client_channel_table.h
        demo_client_channel_table_test.c
demo_client_classes.icl
    demo_client_classes.c
    demo_client_classes.h
demo_client_config.opf
    demo_client_config.icl
        demo_client_config.c
        demo_client_config.h
        demo_client_config_test.c
demo_client_connection.icl
    demo_client_connection.c
    demo_client_connection.h
    demo_client_connection_test.c
demo_client_method.icl
    demo_client_method.c
    demo_client_method.h
    demo_client_method_test.c
demo_client_session.icl
    demo_client_session.c
    demo_client_session.h
    demo_client_session_test.c
```

See base2/asl/project.pdl for the project specification needed for the demo client stack, providing Boom with all information necessary to generate, compile, link, and install the files.

3.5.2 Generating the server stack

To generate the server stack manually we issue this gsl command:

```
gsl demo_server_proto.asl
```

In a PDL project file, this is expressed as:

```
<file name = "demo_server_proto.asl" class = "gsl data" />
```

The generated server stack consists of these files:

- demo_server_agent.smt - the server-side protocol state machine.
- demo_server_method.icl - the server method dispatcher class.

These files are all models, and themselves need to be "compiled" via code generation. When we do this fully, we get this additional set of generated files for the server stack (with .c and .f files generated for classes not shown, for brevity):

```
demo_broker_agent.smt
demo_exchange_agent.smt
demo_exchange_table.icl
demo_queue_agent.smt
demo_queue_table.icl
demo_server_agent.smt
demo_server_channel_agent.smt
demo_server_channel_table.icl
demo_server_config.icl
demo_server_connection_list.icl
demo_server_connection_table.icl
demo_server_method.icl
```

See base2/asl/project.pdl for the project specification needed for the demo server stack, providing Boom with all information necessary to generate, compile, link, and install the files.

3.5.3 Generating the common classes

A number of classes are shared by the client and server stack. These are generated from the protocol base specification (demo.asl). We would issue this gsl command by hand:

```
gsl demo.asl
```

In a PDL project file, this is expressed as:

```
<file name = "demo.asl" class = "gsl public data" />
```

The generated shared classes consist of these files:

- demo_constants.icl - protocol constants.
- demo_content_basic.icl - the shared Basic content class.
- demo_content_basic_list.icl - list of Basic contents.

These are the final files generated when we compile all the models:

```
demo_constants.icl
    demo_constants.c
    demo_constants.h
demo_content_basic.icl
    demo_content_basic.c
    demo_content_basic.h
    demo_content_basic_test.c
demo_content_basic_list.icl
    demo_content_basic_list.c
    demo_content_basic_list.h
    demo_content_basic_list_test.c
```

See base2/asl/project.pdl for the project specification needed for the demo shared clases, providing Boom with all information necessary to generate, compile, link, and install the files.

4 Predefined targets

4.1 The stdc target

The stdc target generates code for an ANSI C client stack, and an ANSI C server stack, both of which are built using iMatix model oriented technology (iCL, etc.)

4.1.1 Overview of client stack

The default client stack provides a synchronous API. That is, the application makes a call, and the stack blocks until the work is completed.

However, the client stack is internally multithreaded and fully asynchronous and executes the protocol in the background.

So we also provide an asynchronous API which works by sending method requests and receiving method responses, without blocking. This is useful for asynchronous applications written using the iMatix Base2 framework.

To generate a synchronous API, the client chassis must specify:

```
<option name = "syncapi" value = "1" />
```

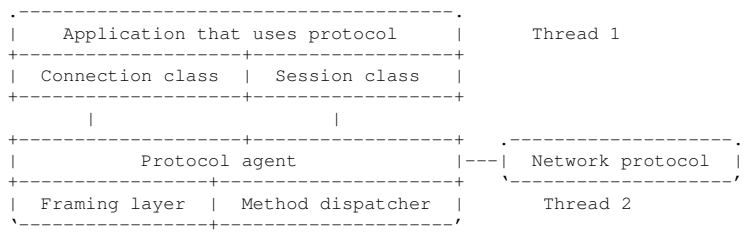
Otherwise, the stdc target generates an asynchronous client stack.

4.1.2 The synchronous client stack

4.1.2.1 General architecture

The synchronous stdc client stack generates a library that consists of these generated layers:

- The connection layer. A connection is an API object that maps onto protocol connections (as defined by the asl_connection class provided in the ASL base specification).
- The session layer. A session is an API object that maps onto individual channels (as defined by the asl_channel class provided in the ASL base specification).
- A protocol agent, which handles the semantics of the protocol methods, from the client point of view.
- A framing layer, which encodes and decodes protocol frames.
- A method dispatcher layer, which processes incoming protocol methods.



Synchronisation works as follows:

- The application invokes function calls (synchronous methods) on either a connection or session object.
- Each connection and session object has their own method queue.

- These object pass those calls to the protocol agent via a method queue, the standard SMT technique for sending requests to an async agent.
- The connection or session object waits for a response on their own method queue. This blocks the
- The protocol agent receives and executes these methods in due time, following its own state machine execution.
- The protocol agent, when necessary, sends reply methods back to the connection or session object that made the request.

The protocol agent can actually send methods back to an open session at any time; this is specifically used to carry messages (contents) to the API.

An operation like "wait for message" in the session then becomes:

- Inspect session method queue, remove any messages.
- If there were none, wait until some arrive.

4.1.2.2 The connection class

A connection object establishes and manages a single connection to a protocol server. The connection can hold many sessions - each is a session object. To use the API, we first create a connection to the server, then we create one or more sessions and do work on the sessions. Each session represents a serial stream of work. In simple applications, a single session is usually sufficient. In multithreaded applications, each thread will want its own session.

For a detailed understanding of the connection class, read the file demo_client_connection.icl. You will need a good knowledge of ANSI C, and iCL models and techniques to understand this file.

4.1.2.3 The session class

A session object implements a protocol channel. The session class provides the main protocol API, with one API method generated for each outgoing protocol method.

For a detailed understanding of the session class, read the file demo_client_session.icl. You will need a good knowledge of ANSI C, and iCL models and techniques to understand this file.

4.1.2.4 The protocol agent

The protocol agent is implemented as an SMT state machine. It has a single thread per connection, and manages the channels within the connection thread using a hash structure (a table container).

Each thread progresses through a set of states, each state accepts a well-defined set of events (from thread actions) and/or methods (from the session and/or connection objects associated with that connection thread).

The methods that the agent accepts are:

- channel open - open a new channel.
- channel close - close a specified channel.
- connection close - close the current connection.

class_name [method_name] - send protocol method to server.

- push - send a formatted protocol method frame to the server.

The main states that the agent implements are:

- initialise connection - opens the connection and handles the whole connection negotiation sub-protocol.
- connection active - the main state, in which we handle either a method from the session/connection objects, or incoming frames from the server. When we get incoming frames from the server this state passes control to a "have_" state to process the frame.
- have method class - in this state we process the different classes of method frames incoming from the server. Note that connection and channel are predefined in the stdc target specification, and the other classes are taken from the protocol specification.
- have connection method - this state handles the different connection methods incoming from the server.
- have channel method - this state handles the different channel methods incoming from the server.
- defaults - the defaults state handles events and methods that are not explicitly handled in the current thread state. This is used both to simplify the state machine (by centralising event handling) and for error handling (to trap unexpected events).

Additionally the agent implements a number of sub-states, which are called from other states to perform some specific work:

- read non heartbeat frame - read a non-heartbeat frame from the connection, return when we have a non-heartbeat frame.
- read [class.name] content - this state reads a content sent from the server, frame by frame, return when the content is complete.
- active close - close the connection using the handshake model defined in the connection sub-protocol.

The client protocol agent has these known limitations:

- It cannot handle contents sent on multiple channels at once. The "read [class.name] content" state does not allow this.
- It serialises network input and output. For very high-volume scenarios it would be more efficient to use two threads per connection, one for input and one for output. The server agent implements such a model. (Note that this is also significantly more complex).

Error handling is done in the defaults state, which traps all unexpected events, socket errors, timeouts, and heartbeat failures, and handles them as needed. Most errors result in a simple closure of the socket, and destruction of connection resources. The "connection error" event, which is sent by the server when it detects a protocol error, uses the more pedantic handshaken closure of the "active close" state.

For a detailed understanding of the protocol agent, read the file demo_client_agent.smt. You must have a good knowledge of SMT models and techniques, since this is a fairly complex piece of machinery.

4.1.2.5 The basename symbol

The stdc target uses a single symbol, "basename", as a prefix for function names and filenames.

This symbol can be defined in any hand-written class included into the client stack. By convention we define it in the client channel class. For example:

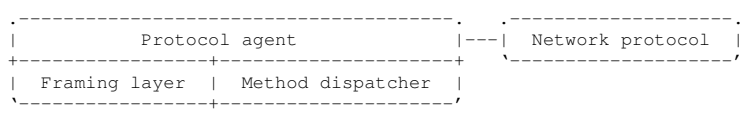
```
<class
    name    = "demo_client_channel"
    comment = "Demo client channel class"
    script  = "icl_gen">
<doc>
This class implements the ASL channel class for the demo client.
</doc>
<inherit class = "asl_client_channel" />
<option name = "basename" value = "demo_client" />
</class>
```

4.1.3 The asynchronous client stack

4.1.3.1 General architecture

The asynchronous stdc client stack generates a library that consists of these generated layers:

- A protocol agent, which handles the semantics of the protocol methods, from the client point of view.
- A framing layer, which encodes and decodes protocol frames.
- A method dispatcher layer, which processes incoming protocol methods.



This is very similar to the bottom half of the synchronous stack; the connection and session layer in the synchronous stack in fact work to provide synchronisation, and are not needed in the async architecture.

The protocol agent is also generated differently, since it does not respond to a session/connection but rather to some other application object.

The async client stack is really intended to be embedded in a fully async application - a server, usually - written using the iMatix Base2 framework.

4.1.3.2 The protocol agent

The async client stack protocol agent has the same overall design as the synchronous agent, minus some synchronisation with the session layer. If you read the asl_client_agent.gsl target generator, these differences are clear, e.g.:

```
.if defined (syncapi)
    <action name = "reply connection ready" />
.else
    <action name = "wait for activity" />
.endif
```

Where the synchronous version tells the session layer that it is ready for a method, while the async version simply waits for method or socket activity.

4.1.3.3 The basename symbol

For the async client stack, which has no client_channel class, the basename symbol can be specified in the protocol chassis, e.g.:

```
<protocol
    comment  = "Embedded protocol client"
    script   = "asl_gen"
    chassis  = "client"
    basename = "my_proxy"
    >
```

4.1.3.4 Hand-written files

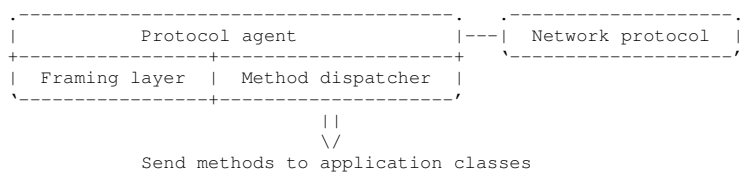
The async client stack does not need the same hand-written files as the synchronous stack. The only hand-written file that is needed is basename_config.opf.

4.1.4 The server stack

4.1.4.1 General architecture

The server stack is a framework that handles the protocol on behalf of a set of server classes. These are the generated layers in that framework:

- A protocol agent, which handles the semantics of the protocol methods, from the client point of view.
- A framing layer, which encodes and decodes protocol frames.
- A method dispatcher layer, which processes incoming protocol methods.



4.1.4.2 Server protocol state machine

The protocol agent is implemented as an SMT state machine. It has a one master thread, and two threads per connection, one for input and one for output, and manages the channels within these threads using a hash structure (a table container).

The agent has these thread types:

- master - this thread handles socket connection requests.
- connection - this thread handles connection start-up, and output.
- input - this thread handles socket input.

Each thread progresses through a set of states, each state accepts a well-defined set of events (from thread actions) and/or methods (from application objects).

The methods that the agent accepts are:

- set trace - set trace level on a specific connection.
- kill connection - kill a specific connection.
- connection open ok - send Connection.Open-Ok to the client.
- connection redirect - send Connection.Redirect to the client.

class_name [method_name] - send outgoing protocol method to client.

- push - send a formatted protocol method frame to the client.

The master thread monitors the network socket for new connections. This thread has the following states:

- initialise master - open the socket and accept incoming connection requests.
- new connection - create a new connection thread, and accept another connection.

The connection thread does connection start-up negotiation, and handles outgoing methods. It has these states:

- initialise connection - opens the connection and handles the whole connection negotiation sub-protocol.
- connection active - the main state, in which we handle methods from application objects.

The input thread waits for socket input and handles incoming protocol frames, in parallel with the connection thread, so that input and output can happen simultaneously. It has these states:

- start - this state starts the input cycle by reading the first frame.
- have method class - in this state we process the different classes of method frames incoming from the server. Note that connection and channel are predefined in the stdc target specification, and the other classes are taken from the protocol specification.
- have connection method - this state handles the different connection methods incoming from the server.
- have channel method - this state handles the different channel methods incoming from the server.

Default states are used at two levels: in the connection and input threads, and also at the top level of the state machine.

Additionally the connection and input threads use a number of sub-states that are called from other states to perform some specific work:

- read non heartbeat frame - read a non-heartbeat frame from the connection, return when we have a non-heartbeat frame.
- read [class.name] content - this state reads a content sent from the server, frame by frame, return when the content is complete.
- active close - close the connection using the handshake model defined in the connection sub-protocol.

Error handling is done in the defaults states, which trap all unexpected events, socket errors, timeouts, and heartbeat failures, and handles them as needed. Most errors result in a simple closure of the socket, and destruction of connection resources. The "connection error" event, which is sent by the server when it detects a protocol error, uses the more pedantic handshaken closure of the "active close" state.

For a detailed understanding of the protocol agent, read the file demo_server_agent.smt. You must have a good knowledge of SMT models and techniques, since this is a highly complex piece of machinery.

4.1.4.3 The basename symbol

The stdc target uses a single symbol, "basename", as a prefix for function names and filenames.

This symbol can be defined in any hand-written class included into the client stack. By convention we define it in the server channel class. For example:

```
<class
    name    = "demo_server_channel"
    comment = "Demo server channel class"
    script  = "smt_object_gen">
<doc>
This class implements the ASL demo server channel class.
</doc>
<inherit class = "asl_server_channel" />
<option name = "basename" value = "demo_server" />
</class>
```

4.1.5 Protocol options

A number of options need to be specified in the protocol base specification. These are (with example values):

```
<option name = "protocol_name"     value = "DEMO" />
<option name = "protocol_port"     value = "7654" />
<option name = "protocol_class"    value = "1" />
<option name = "protocol_instance" value = "1" />
<option name = "protocol_major"    value = "1" />
<option name = "protocol_minor"    value = "1" />
```

See demo.asl for an example. The meaning of each of these is:

- protocol_name - the name of the protocol, which acts as the prefix for hand-written files, and some generated files.
- protocol_port - the default port for protocol network connections.
- protocol_class - the 'class' used in version negotiation.
- protocol_instance - the 'instance' used in version negotiation.
- protocol_major - the major version number.
- protocol_minor - the minor version number.

In the client chassis, these options are allowed:

```
<option name = "product_name" value = "ASL Demo Client" />
<option name = "syncapi"      value = "1" />
```

- product_name - a string that is sent to the server at connection time, and used for logging.
- syncapi - if 1, the stdc target generates a synchronous client stack; if 0, or not specified, the stdc target generates an async client stack.

In the server chassis, these options are allowed:

```
<option name = "product_name" value = "ASL Demo Server" />
```

- product_name - a string that is sent to the client at connection time, and used for logging.

4.2 The doc target

The doc target generates detailed documentation for a protocol, from the protocol classes and methods, and the documentation that is embedded in the specification.

To generate documentation for a protocol, use this command (here we use the demo protocol as an example:

```
gsl -target:doc demo.asl
```

In a PDL project specification, use this syntax (replace 'demo.asl' with the name of your protocol base specification):

```
<actions>
  <generate>
    <execute>
    gsl -q -quiet:1 -target:doc demo.asl
    </execute>
  </generate>
</actions>
```

The generated documentation uses the iMatix gurudoc format; this text format is easy to convert into other forms, either using the iMatix gurudoc toolkit, or using simple conversion scripts written in languages like Perl. The base/gurudoc/gd2xhtml script is an example of a stand alone gurudoc-to-XHTML convertor.

The doc target produces these files (for the demo protocol as an example):

- demo_full.txt - full documentation for each class and method.
- demo_quick.txt - summaries of each class and method.
- demo_ids.txt - a reference of class and method numbering (IDs).
- demo_replies.txt - a summary of the protocol reply codes.

4.3 The pal target

PAL is the Protocol Automation Language, and documented elsewhere. PAL is a protocol scripting framework. It works as follows:

- From the protocol specification, the pal target generates a grammar for a protocol automation language that specifically supports the protocol in question. E.g. from demo.asl we get a grammar for a scripting language specifically for the demo protocol.
- From the grammar, PAL generates a compiler for the language; this compiler takes scripts and turns them into source code in target languages - the default is stdc.

The actual code generation process is complex but we can ignore it, and focus on the results, and how to use them.

This is how to generate the scripting grammar and compiler for a protocol (again, we use demo.asl as the example):

```
gsl -target:pal demo.asl
gsl demo_pal.xnf
```

This produces a set of files, the main one being demo_pal_gen.gsl, the main compiler for our protocol scripting language. We can then use this in scripts as follows:

```
<?xml?>
<pal script = "demo_pal_gen">
    This script demonstrates how to do a simple loop using the built-in
    facilities of the repeat command.
    <repeat times = "10" counter = "index">
        <echo>I can count up to $index</echo>
    </repeat>
</pal>
```

5 Base ASL protocol

5.1 Overview

5.1.1 A standard transport layer

ASL is a general-purpose framework for generating code and documentation from protocol specifications.

The only thing that is absolutely fixed in the range of protocols that ASL can implement is that they adopt the class/method semantics enforced by the asl language itself.

However, a large part of ASL's value comes from the provision of a standard transport layer that is capable of implementing arbitrary protocols very rapidly.

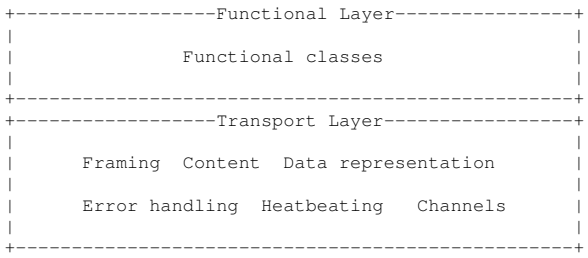
This standard transport layer is implemented in:

- A number of pre-supplied ASL specifications that define the class/method semantics for the transport layer
- The pre-supplied targets, especially stdc, which generate code for the transport layer.

5.1.2 Wire-level protocol model

The ASL transport layer is a binary protocol with modern features: it is multi-channel, negotiated, asynchronous, secure, portable, neutral, and efficient.

We can usefully split any ASL protocol that uses this transport layer into two layers:



The functional layer defines a set of classes and methods that do useful work on behalf of the application.

The transport layer that carries these methods from application to server, and back, and which handles channel multiplexing, framing, content encoding, heartbeating, data representation, and error handling.

One could replace the transport layer with arbitrary transports without changing the application-visible functionality of the protocol. One could also use the same transport layer for different high-level protocols.

The design of the transport layer was driven by these main requirements, in no particular order:

- To be compact, using a binary encoding that packs and unpacks rapidly.
- To handle messages of any size without significant limit.
- To allow zero-copy data transfer (e.g. remote DMA).
- To carry multiple channels across a single connection.
- To be long-lived, with no significant in-built limitations.
- To allow asynchronous command pipelining.

- To be easily extended to handle new and changed needs.
- To be forward compatable with future versions.
- To be repairable, using a strong assertion model.
- To be neutral with respect to programming languages.
- To have no license or patent restrictions.
- To fit a code generation process.

5.1.3 Designing the wire-Level protocol

The wire-level protocol is designed as multiple elements that work together, mainly being:

1. The protocol header: how connections are opened.
2. The framing layer: how data is delimited on the connection.
3. The data types: how data fields are formatted.
4. The method layer: how methods are carried between peers.
5. The content layer: how content is carried between peers.

The protocol header is distinct from the rest of the protocol, and is designed to let the server and client agree on the protocol before doing any real work in terms of frames. Everything that follows the protocol header is built on frames.

5.1.4 The Protocol Header

We wanted to allow a server to support multiple protocols on the same socket. This means we must detect the protocol immediately, before any commands are sent. Most existing protocols expect the client to send a short header (e.g. HTTP, SMTP, FTP, etc.) to signal their intent. We adopted the same convention, and added simple version negotiation so that clients and servers can agree on their compatability before doing any work.

The protocol header mechanism works like this:

- The client opens a socket and writes a protocol header.
- The server "sniffs" the first four characters and checks that it understands this.
- The server then reads the next octets, which indicate the version that the client wishes to use.
- The server either accepts this, or writes its own protocol header and closes the socket.

The protocol negotiation design lets us add new protocol variations in a clean fashion. For example, we might define a new "ultra-compact" framing mechanism; this could co-exist with the exsting framing mechanism, as a different protocol instance within the same family. It is somewhat esoteric, but plausible enough to be worth supporting.

5.1.5 The Framing Layer

We wanted a mechanism that would be fast, long-lived, and support multiple channels on one connection.

The first choice is how to delimit frames. There are several ways to do this, the most efficient is to write the size of the frame, and then write the frame. The reader picks up the size, and then uses that to read the frame. It is fast and safe: it makes it impossible for a peer to block another by sending oversized frames.

Our frame consists of a short frame header, a frame payload, and a frame end octet. As well as defining the frame size, this mechanism solves a number of other issues:

- The frame header holds a frame type, letting us separate different types of frame at the most basic level. For instance, commands (we call these "methods") have a different frame type from data (we call this "content").
- The frame header holds a channel number, which lets the peers send frames for multiple channels across a single connection.
- The frame end octet lets us trap malformed frames, a typical problem with newly-written client code.

The framing mechanism is easy to extend with new frame types, and easy to improve, since each frame type is well separated (changing the way data is formatted has no impact on the way commands are formatted, for example).

5.1.6 The Data Types

We want a uniform data repesentation that would be portable, compact, safe, long-lived, and language neutral. We need to be able to represent those data that the server actually manipulates, which are essentially:

- Bits, used for indicators.
- Numbers, used to indicate quantities and sizes.
- Strings, used for routing and matching.
- Tables, used to hold optional fields.

Everything that peers send using the transport layer is composed of these data types.

In our structures, we pack data down to the octet level. This means that structures cannot be read directly into memory areas. However, integers must in any case be converted to/from network byte order, and large fields are formatted as length-specified binary strings, so there is usually not a significant extra overhead in unpacking data octet by octet where needed.

5.1.6.1 Bit Data Type

The bit data type represents yes/no options, which are fairly common in the protocol. To save space, bits are packed into octets.

5.1.6.2 Number Data Types

Numbers are unsigned integers (there is no case where we need signed values), and we use 8-bit, 16-bit, 32-bit, and 64-bit precision as needed. These are called "octet", "short", "long", and "longlong" respectively.

5.1.6.3 String Data Types

Strings are either:

1. Short printable strings intended for consumption by the server, stored as octet+content, up to 255 characters long, and without any null octets.
2. Long binary strings, intended to carry opaque or encoded content, stored as long+content, up to 4Gb octets long.

These two formats solve a number of issues. First, the short string format is safe, fast, and clear. There are no user-specified strings in the protocol that are larger than 255 octets. Second, both string formats are fast to read and write, since we can copy them in bulk.

5.1.6.4 Field Table Data Type

The field table data type is encoded as a long string but internally it holds a series of fields, specified as name, type, and value. Field tables hold data at the application level - e.g. message headers - so these fields have slightly different types from simple fields. Integers are signed, there is support for decimal values (e.g. to hold exchange rates).

Field tables are useful wherever we need a variable set of named arguments, rather than an explicitly named set of arguments. We do not pass all data as field tables because it makes the protocol less explicit and more verbose to work with.

5.1.7 The Method Layer

By carrying all commands as method frames, we create a good framework for adding and extending the command set that the protocol provides. Indeed, the same framing mechanism can be used for arbitrary command sets. Making reusable technology is always better.

We wanted the method layer to have these characteristics:

- Support fully asynchronous operation (which is fast) as well as synchronous operation (which is simple), on a case-by-case basis.
- Be easily extensible so that we can create many methods, each doing one thing, rather than a few large and complex methods. It is much easier to implement and verify a large set of small single-function methods than a small set of large multi-function methods.

Asynchronicity is very important in a network protocol. The cost of acknowledging a command is significant: a TCP/IP round trip can take a full second unless one switches off all buffering (the Nagle option). An asynchronous command implies no round trip: the sender can push such commands at full speed towards the recipient, and the commands will be queued as close to the recipient as possible.

The protocol has features specifically designed for asynchronicity:

- Explicit definition of methods as "synchronous" or "asynchronous".
- An error-handling mechanism (exceptions) that supports a fully asynchronous dialogue.

To make the command set extensible, we use a class-method semantic. The command set refers to a set of classes, each covering one functional domain. We have four distinct types of class:

- Classes that manage the transport process (Connection and Session).
- Classes that manage the workflow (Access, Transaction).
- Classes that represent the main AMQ entities (Queue, Exchange).
- Classes that represent functional domains (Basic, File, Stream).

For each class we define a set of methods that:

- Are named.
- Have a set of named arguments.
- Are either synchronous requests, or replies, or asynchronous.
- Are defined as being server-side and/or client-side.
- Are defined as MUST, SHOULD, MAY implement.

The protocol starts to look just like RPC, with one major difference: we allow fully asynchronous operations.

5.1.8 The Content Layer

Content is message data, consisting of a set of properties, and a block of binary data. The challenges for content transfer are:

- Ability to handle any size of content, efficiently. This includes the fair mixing of large and small contents across multiple channels in a single connection.
- To allow zero-copy data transfer (e.g. remote DMA).
- To allow structured (multi-part) data. For instance, a video clip may consist of some XML metadata followed by several video segments.
- To be very compact for simple cases, which includes empty contents. Specifically, the cost of carrying empty properties must be low.

Our design is this:

- All content consists of a header plus a body. The header holds the content properties, and the body holds the content data.
- The body can be from 0 to 16Eb large. Large contents are split into frames that are sent as a series. Frames for different channels are mixed within the connection, so smaller contents can be sent at the same time as larger contents, on different channels.
- Each body frame consists of a frame header, payload and frame-end as for all frames. The payload can be sent out-of-band, so that the actual content transfer on the connection is minimal.
- Contents can be structured in a tree form, each content header being followed by child contents, to any level.

The nice thing about this design is that it's simple for simple cases but scalable to very complex cases. Although the maximum size of a single content is 16 exabytes (2E64), the use of multipart contents means that the combined content can be up to 1 yottabyte (2E80) if no nesting is used, and using nesting, there is no limit.

5.1.9 The Error Handling Model

The protocol error handling model is designed to be simple, robust, and unambiguous.

Most methods have a single possible response. Some methods have multiple responses. The most heavily used methods are entirely asynchronous and do not expect responses.

When a method succeeds, the server responds if the method is synchronous, or says nothing if the method is asynchronous.

When a method fails, the server "raises an exception", which is a fatal error. The exception can happen at two levels:

- The channel level, for soft errors (configuration, usually).
- The connection level, for hard errors (programming faults, usually).

After an exception, both sides close the channel or connection using a hand-shaked procedure. Any work in progress is discarded.

The advantages of this model are that:

1. Success is silent, in the case of asynchronous methods, so the protocol is not chatty, and fast.
2. Failure is unrecoverable, so errors get fixed. I.e. any fault is highlighted very rapidly, and must be fixed.

The technique of "clear failure" is widely used to force applications to be robust. This is part of the reasoning behind the use of assertions in programming.

5.2 Protocol implementation

5.2.1 The class/method model

Protocol commands are grouped into 'classes', each class covering a specific area. Classes are implemented as a set of 'methods'.

There are two distinct method dialogues:

- Synchronous request-response, in which one peer sends a request and the other peer sends a reply. Synchronous request and response methods are used for functionality that is not performance critical.
- Asynchronous notification, in which one peer sends a method but expects no reply. Asynchronous methods are used where where performance is critical.

To make method processing simple, we define distinct replies for each synchronous request. That is, no method is used as the reply for two different requests. This means that a peer, sending a synchronous request, can accept and process incoming methods until getting one of the valid synchronous replies.

A method is formally defined as a synchronous request, a synchronous reply (to a specific request), or asynchronous. Lastly, each method is formally defined as being client-side (i.e. server to client), or server-side (client to server).

5.2.2 No confirmations

A chatty protocol is slow. We use asynchronicity heavily in those cases where performance is an issue. This is generally where we send content from one peer to another. We send off methods as fast as possible, without waiting for confirmations. Where necessary, we implement windowing and throttling at a higher level, e.g. at the consumer level.

We can dispense with confirmations because we adopt an assertion model for all actions. Either they succeed, or we have an exception that closes the channel or connection.

There are no confirmations in the protocol. Success is silent, and failure is noisy. When applications need explicit tracking of success and failure, they can define transaction semantics as functional classes.

5.2.3 The connection class

The protocol is connected. The connection is designed to be long-lasting, and can carry multiple channels. The connection life-cycle is this:

- The client opens a TCP/IP connection to the server and sends a protocol header. This is the only data the client sends that is not formatted as a method.
- The server responds with its protocol version and other properties, including a list of the security mechanisms that it supports (the Start method).
- The client selects a security mechanism (Start-Ok).
- The server starts the authentication process, which uses the SASL challenge-response model. It sends the client a challenge (Secure).
- The client sends an authentication response (Secure-Ok). For example using the "plain" mechanism, the response consist of a login name and password.

- The server repeats the challenge (Secure) or moves to negotiation, sending a set of parameters such as maximum frame size (Tune).
- The client accepts or lowers these parameters (Tune-Ok).
- The client formally opens the connection and selects a virtual host (Open).
- The server confirms that the virtual host is a valid choice (Open-Ok).
- The client now uses the connection as desired.
- One peer (client or server) ends the connection (Close).
- The other peer hand-shakes the connection end (Close-Ok).
- The server and the client close their socket connection.

5.2.4 The channel class

The protocol is multi-channeled. Channels are independent threads of control that share a connection. The channel life-cycle is this:

- The client opens a new channel (Open).
- The server confirms that the new channel is ready (Open-Ok).
- The client and server use the channel as desired.
- One peer (client or server) closes the channel (Close).
- The other peer hand-shakes the channel close (Close-Ok).

5.3 The transport layer

5.3.1 General description

The standard transport layer is a binary protocol. Information is organised into "frames", of various types. Frames carry protocol methods, structured contents, and other information. All frames have the same general format: frame header, payload, and frame end. The frame payload format depends on the frame type.

We assume a reliable stream-oriented network transport layer (TCP/IP or equivalent). However, the transport layer can be implemented over other networking protocols such as SCTP, and PGM multicast.

Within a single socket connection, there can be multiple independent threads of control, called "channels". Each frame is numbered with a channel number. By interleaving their frames, different channels share the connection. For any given channel, frames run in a strict sequence that can be used to drive a protocol parser (typically a state machine).

We construct frames using a small set of data types such as bits, integers, strings, and field tables. Frame fields are packed tightly without making them slow or complex to parse. It is relatively simple to generate framing layer mechanically from the protocol specifications.

5.3.2 Data types

The standard data types are:

- Integers (from 1 to 8 octets), used to represent sizes, quantities, limits, etc. Integers are always unsigned and may be unaligned within the frame.
- Bits, used to represent on/off values. Bits are packed into octets.

- Short strings, used to hold short text properties. Short strings are limited to 255 octets and can be parsed with no risk of buffer overflows.
- Long strings, used to hold chunks of binary data.
- Field tables, which hold name-value pairs. The field values are typed as strings, integers, etc.

5.3.3 Protocol negotiation

The client and server negotiate the protocol. This means that when the client connects, the server proposes certain options that the client can accept, or modify. When both peers agree on the outcome, the connection goes ahead. Negotiation is a useful technique because it lets us assert assumptions and preconditions.

We negotiate a number of specific aspects of the protocol:

1. The actual protocol and version. A server can host multiple protocols on the same port.
2. Encryption arguments and the authentication of both parties. This is part of the functional layer, explained previously.
3. Maximum frame size, number of channels, and other operational limits.

Agreed limits allow both parties to pre-allocate key buffers, avoiding deadlocks. Every incoming frame either obeys the agreed limits, so is "safe", or exceeds them, in which case the other party is faulty and can be disconnected. Both peers negotiate the limits to the lowest agreed value as follows:

1. The server tells the client what limits it proposes.
2. The client can respond to lower the limits for its connection.

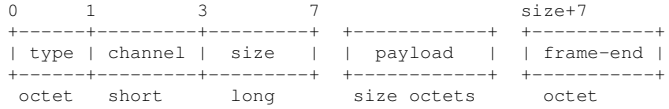
5.3.4 Delimiting frames

TCP/IP is a stream protocol, i.e. there is no in-built mechanism for delimiting frames. Existing protocols solve this in several different ways:

- Sending a single frame per connection. This is simple but slow.
- Adding frame delimiters to the stream. This is simple but slow to parse.
- Counting the size of frames and sending the size in front of each frame. This is simple and fast, and our choice.

5.3.5 Frame details

All frames consist of a header (7 octets), a payload of arbitrary size, and a 'frame-end' octet that detects malformed frames:



To read a frame, we:

1. Read the header and check the frame type and channel.
2. Depending on the frame type, we read the payload and process it.
3. Read the frame end octet.

In realistic implementations where performance is a concern, we would use read-ahead buffering to avoid doing three separate system calls to read a frame.

5.3.5.1 Method frames

Method frames carry the high-level protocol commands (which we call "methods"). One method frame carries one command. The method frame payload has this format:

```
0          2           4
+----------+-----------+-------------- - -
| class-id | method-id | arguments...
+----------+-----------+-------------- - -
   short      short     ...
```

To process a method frame, we:

1. Read the method frame payload.
2. Unpack it into a structure. A given method always has the same structure, so we can unpack the method rapidly.
3. Check that the method is allowed in the current context.
4. Check that the method arguments are valid.
5. Pass the method to another layer for actual processing.

Method frame bodies are constructed as a list of data fields (bits, integers, strings and string tables). The marshalling code is trivially generated directly from the protocol specifications, and can be very rapid.

5.3.5.2 Content frames

Content is the application data we carry from box to box. Content is, roughly speaking, a set of properties plus a binary data part. The set of allowed properties are defined by the content class, and these form the "content header frame". The data can be any size, and can be broken into several (or many) chunks, each forming a "content body frame".

Looking at the frames for a specific channel, as they pass on the wire, we might see something like this:

```
[method]
[method] [header] [body] [body]
[method]
...
```

Certain methods (such as Basic.Publish, Basic.Deliver, etc.) are formally defined as carrying content. When a peer sends such a method frame, it always follows it with a content header and zero or more content body frames.

A content header frame has this format:

```
0          2        4           12               14
+----------+--------+-----------+----------------+------------- - -
| class-id | weight | body size | property flags | property list...
+----------+--------+-----------+----------------+------------- - -
   short     short    long long       short        remainder...
```

We place content body in distinct frames (rather than including it in the method) so that we support "zero copy" techniques in which content is never marshalled or encoded, and can be sent via out-of-band transport such as shared memory or remote DMA.

We place the content properties in their own frame so that recipients can selectively discard contents they do not want to process.

Contents can be structured with sub-contents to any level.

5.3.5.3 Out-of-band frames

Out-of-band transport can be used in specific high-performance models. Note that this part of the protocol is speculative because we have not built a working out-of-band prototype. This part of the protocol is a placeholder rather than a formal proposal.

The principle of out-of-band transport is that a TCP/IP connection can be used for controlling another, faster but less abstract protocol such as remote-DMA, shared memory, or multicast.

5.3.5.4 Heartbeat frames

Hearbeating is a technique designed to undo one of TCP/IP's features, namely its ability to recover from a broken physical connection by closing only after a quite long timeout. In some scenarios we need to know very rapidly if a peer is disconnected or not responding for other reasons (e.g. it is looping). Since heartbeating can be done at a low level, we implement this as a special type of frame that peers exchange at the transport level, rather than as a class method.

5.3.6 Error handling

We use exceptions to handle errors. That is:

- Any operational error, e.g. message queue not found, insufficient access rights, etc. results in a channel exception.
- Any structural error, e.g. invalid argument, bad sequence of methods, etc. results in a connection exception.

An exception closes the channel or connection, and returns a reply code and reply text to the client application. We use the 3-digit reply code plus textual reply text scheme that is used in HTTP and many other protocols.

5.3.7 Closing channels and connections

Closing a channel or connection for any reason - normal or exceptional - must be done carefully. Abrupt closure is not always detected rapidly, and following an exception, we could lose the error reply codes. The correct design is to hand-shake all closure so that we close only after we are sure the other party is aware of the situation.

When a peer decides to close a channel or connection, it sends a Close method. The receiving peer responds with Close-Ok, and then both parties can close their channel or connection.

5.4 Wire-Level Format

5.4.1 Format Protocol Grammar

We provide a complete protocol grammar for (this is provided for reference, and you may find it more interesting to skip through to the next sections that detail the different frame types and their formats):

```
protocol            = protocol-header *unit
protocol-header     = literal-ID protocol-id protocol-version
literal-ID          = 4*CHAR                    ; "NAME"
protocol-id         = 2*OCTET
protocol-version    = 2*OCTET
unit                = method | oob-method | trace | heartbeat
method              = method-frame [ content ]
method-frame        = %d1 frame-properties method-payload frame-end
frame-properties    = channel payload-size
channel             = short-integer             ; Non-zero
payload-size        = long-integer
method-payload      = class-id method-id *field
class-id            = %x00.01-%xFF.FF
method-id           = %x00.01-%xFF.FF
field               = BIT / OCTET
                    / short-integer / long-integer / long-long-integer
                    / short-string / long-string
                    / timestamp
                    / field-table
short-integer       = 2*OCTET
long-integer        = 4*OCTET
long-long-integer   = 8*OCTET
short-string        = OCTET *string-char        ; length + content
string-char         = %x01 .. %xFF
long-string         = long-integer *OCTET       ; length + content
timestamp           = long-long-integer
field-table         = long-integer *field-value-pair
field-value-pair    = field-name field-value
field-name          = short-string
field-value         = 'S' long-string
                    / 'I' signed-integer
                    / 'D' decimal-value
                    / 'T' timestamp
                    / 'F' field-table
signed-integer      = 4*OCTET
decimal-value       = decimals long-integer
decimals            = OCTET
frame-end           = %xCE
content             = %d2 content-header child-content *content-body
content-header      = frame-properties header-payload frame-end
header-payload      = content-class content-weight content-body-size
                      property-flags property-list
content-class       = OCTET
content-weight      = OCTET
content-body-size   = long-long-integer
property-flags      = 15*BIT %b0 / 15*BIT %b1 property-flags
property-list       = field
child-content       = content-weight*content
content-body        = %d3 frame-properties body-payload frame-end
body-payload        = *OCTET
oob-method          = oob-method-frame [ oob-content ]
oob-method-frame    = %d4 frame-properties frame-end
oob-content         = %d5 content-header oob-child-content
                      *oob-content-body
oob-child-content   = content-weight*oob-content
oob-content-body    = %d6 frame-properties frame-end
trace               = %d7 frame-flags %d0 payload-size trace-payload
                      frame-end
trace-payload       = *OCTET
heartbeat           = %d8 frame-flags %d0 %d0 frame-end
```

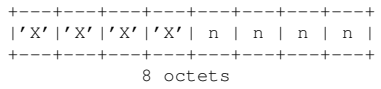
We use the Augmented BNF syntax defined in IETF RFC 2234. In summary,

- The name of a rule is simply the name itself.
- Terminals are specified by one or more numeric characters with the base interpretation of those characters indicated as 'd' or 'x'.

- A rule can define a simple, ordered string of values by listing a sequence of rule names.
- A range of alternative numeric values can be specified compactly, using dash ("-") to indicate the range of alternative values.
- Elements enclosed in parentheses are treated as a single element, whose contents are strictly ordered.
- Elements separated by forward slash ("/") are alternatives.
- The operator "*" preceding an element indicates repetition. The full form is: "<a>*<b>element", where <a> and <b> are optional decimal values, indicating at least <a> and at most <b> occurrences of element.
- A rule of the form: "<n>element" is equivalent to <n>*<n>element.
- Square brackets enclose an optional element sequence.

5.4.2 Version Negotiation Subprotocol

Version negotiation is a separate sub-protocol. The client MUST start a new connection by sending a protocol header. This is an 8-octet sequence:



The protocol header consists of four uppercase letters followed by:

1. The protocol class.
2. The protocol instance.
3. The protocol major version.
4. The protocol minor version.

The protocol negotiation model is compatible with existing protocols such as HTTP that initiate a connection with an constant text string, and with firewalls that sniff the start of a protocol in order to decide what rules to apply to it.

A client and server agree on a protocol and version as follows:

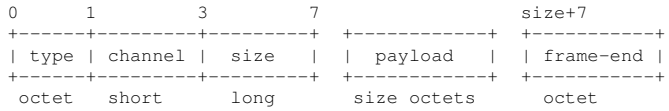
- The client opens a new socket connection to the server and sends the protocol header.
- The server either accepts or rejects the protocol header. If it rejects the protocol header writes a valid protocol header to the socket and then closes the socket.
- Otherwise it leaves the socket open and implements the protocol accordingly.

Guidelines for implementors:

- If the server does not recognise the first 4 octets of data on the socket, or does not support the specific protocol version that the client requests, it MUST write a valid protocol header to the socket, then flush the socket (to ensure the client application will receive the data) and then close the socket connection. The server MAY print a diagnostic message to assist debugging.
- A client MAY detect the server protocol version by attempting to connect with its highest supported version and reconnecting with a lower version if it receives such information back from the server.

5.4.3 General Frame Format

All frames start with an 8-octet header composed of a type field (octet), a frame-flags field (octet), a channel field (short integer) and a size field (long integer):



We define these frame types:

- Type = 1, "METHOD": method frame.
- Type = 2, "HEADER": content header frame.
- Type = 3, "BODY": content body frame.
- Type = 4, "OOB-METHOD": out-of-band method frame.
- Type = 5, "OOB-HEADER": out-of-band band header frame.
- Type = 6, "OOB-BODY": out-of-band body frame.
- Type = 7, "TRACE": trace frame.
- Type = 8, "HEARTBEAT": heartbeat frame.

The channel number is 0 for all frames which are global to the connection and 1-65535 for frames that refer to specific channels.

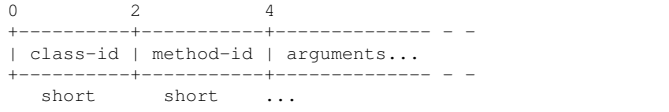
The size field is the size of the payload, excluding the frame-end octet. While we assume a reliable connected protocol, we use the frame end to detect framing errors caused by incorrect client or server implementations.

Guidelines for implementors:

- If a peer receives a frame with a type that is not one of these defined types, it MUST treat this as a fatal protocol error and close the connection without sending any further data on it.
- When a peer reads a frame it MUST check that the frame-end is valid before attempting to decode the frame. If the frame-end is not valid it MUST treat this as a fatal protocol error and close the connection without sending any further data on it. It SHOULD log information about the problem, since this indicates an error in either the server or client framing code implementation.
- A peer MUST NOT send frames larger than the agreed-upon size. A peer that receives an oversized frame MUST signal a connection exception with reply code 501 (frame error).

5.4.4 Method Frames

Method frame bodies consist of an invariant list of data fields, called "arguments". All method bodies start with identifier numbers for the class and method:



Guidelines for implementors:

- The class-id and method-id are constants that are defined in the class and method specifications.
- The arguments are a set of fields that specific to each method.
- Class id values from %x00.01-%xEF.FF are reserved for standard classes.

- Class id values from %xF0.00-%xFF.FF (%d61440-%d65535) may be used by implementations for non-standard extension classes.

5.4.5 Data Fields

5.4.5.1 Integers

We defines these integer types:

- Unsigned octet (8 bits).
- Unsigned short integers (16 bits).
- Unsigned long integers (32 bits).
- Unsigned long long integers (64 bits).

Integers and string lengths are always unsigned and held in network byte order. We make no attempt to optimise the case when two low-high systems (e.g. two Intel CPUs) talk to each other.

Guidelines for implementors:

- Implementors MUST NOT assume that integers encoded in a frame are aligned on memory word boundaries.

5.4.5.2 Bits

Bits are accumulated into whole octets. When two or more bits are contiguous in a frame these will be packed into one or more octets, starting from the low bit in each octet. There is no requirement that all the bit values in a frame be contiguous, but this is generally done to minimise frame sizes.

5.4.5.3 Strings

Strings are variable length and represented by an integer length followed by zero or more octets of data. We define two string types:

- Short strings, stored as an 8-bit unsigned integer length followed by zero or more octets of data. Short strings can carry UTF-8 data, but may not contain binary zero octets.
- Long strings, stored as a 32-bit unsigned integer length followed by zero or more octets of data. Long strings can contain any data.

5.4.5.4 Timestamps

Time stamps are held in the 64-bit POSIX time_t format with an accuracy of one second. By using 64 bits we avoid future wraparound issues associated with 31-bit and 32-bit time_t values.

5.4.5.5 Field Tables

Field tables are long strings that contain packed name-value pairs. Each name-value pair is a structure that provides a field name, a field type, and a field value. A field can hold a tiny text string, a long string, a long signed integer, a decimal, a date and/or time, or another field table.

Guidelines for implementors:

- Field names MUST start with a letter, '$' or '#' and may continue with letters, '$' or '#', digits, or underlines, to a maximum length of 128 characters.

- The server SHOULD validate field names and upon receiving an invalid field name, it SHOULD signal a connection exception with reply code 503 (syntax error). Conformance test: amq_wlp_table_01.
- Specifically and only in field tables, integer values are signed (31 bits plus sign bit).
- Decimal values are not intended to support floating point values, but rather business values such as currency rates and amounts. The 'decimals' octet is not signed.
- A peer MUST handle duplicate fields by using only the first instance.

5.4.6 Content Framing

Certain specific methods (Publish, Deliver, etc.) carry content. Please refer to the chapter "Functional Specifications" for specifications of each method, and whether or not the method carries content. Methods that carry content do so unconditionally.

Content consists of a list of 1 or more frames as follows:

1. Exactly one content header frame that provides properties for the content.
2. Optionally, one or more child contents. A child content follows the exact rules for a content. Contents may thus be structured in a hierarchy to any level.
3. Optionally, one or more content body frames.

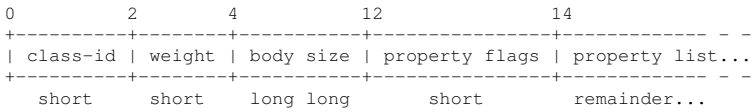
Content frames on a specific channel form an strict list. That is, they may be mixed with frames for different channels, but two contents may not be mixed or overlapped on a single channel, nor may content frames for a single content be mixed with method frames on the same channel.

Guidelines for implementors:

- A peer that receives an incomplete content MUST raise a connection exception with reply code 501 (frame error). Conformance test: amq_wlp_content_01.

5.4.6.1 The Content Header

A content header payload has this format:



Guidelines for implementors:

- The content class-id MUST match the method frame class id. The peer MUST respond to an invalid content class-id by raising a connection exception with reply code 501 (frame error). Conformance test: amq_wlp_content_02.
- The weight field specifies the number of child-contents that the content contains. This is zero for simple contents and non-zero for structured contents (explained below).
- The body size is a 64-bit value that defines the total size of the content body. It may be zero, indicating that there will be no content body frames.
- The property flags are an array of bits that indicate the presence or absence of each property value in sequence. The bits are ordered from most high to low - bit 15 indicates the first property.
- The property flags can specify more than 16 properties. If the last bit (0) is set, this indicates that a further property flags field follows. There are many property flags fields as needed.
- The property values are class-specific data fields.

- Bit properties are indicated ONLY by their respective property flag (1 or 0) and are never present in the property list.
- The channel number in content frames MUST NOT be zero. A peer that receives a zero channel number in a content frame MUST signal a connection exception with reply code 504 (channel error). Conformance test: amq_wlp_content_03.

5.4.6.2 The Content Body

The content body payload is an opaque binary block followed by a frame end octet:

```
+-----------------------+ +------------+
| Opaque binary payload | | frame-end |
+-----------------------+ +------------+
```

The content body can be split into as many frames as needed. The maximum size of the frame payload is agreed upon by both peers during connection negotiation.

Guidelines for implementors:

- A peer MUST handle a content body that is split into multiple frames by storing these frames as a single set, and either retransmitting them as-is, broken into smaller frames, or concatenated into a single block for delivery to an application.

5.4.6.3 Structured Content

A structured content consists of a single top level content and multiple child contents, as complex as needed by the application. Structured contents form a hierarchy, a tree with a single root.

At any level of this tree, the weight field in the content header indicates whether the content has child contents or not. If the content has child contents, these follow immediately after the header and before the body of the parent content:

```
[parent-header weight = 1]
    [child-header weight = 0] [child-body]
[parent-body]
```

Guidelines for implementors:

- The peer MAY support structured contents. If it does not support structured contents it MUST respond to a structured content by raising a connection exception with reply code 540 (not implemented). Conformance test: amq_wlp_content_04.
- The peer MUST correctly detect a mismatch between the content weight and the frames that follow, and report such a mismatch by raising a connection exception with reply code 501 (frame error). Conformance test: amq_wlp_content_05.

5.4.7 Out-Of-Band Frames

The formatting of out-of-band frames follows the same specifications as for normal frames, with the exception that frame payloads are sent via some unspecified transport mechanism. This could be shared memory, specialised network protocols, etc.

The actual out-of-band transport used, and its configuration, is specified in the Channel.Open method.

5.4.8 Trace Frames

Trace frames are intended for a "trace handler" embedded in the recipient peer. The significance and implementation of the trace handler is implementation-defined.

Guidelines for implementors:

- Trace frames MUST have a channel number of zero. A peer that receives an invalid trace frame MUST raise a connection exception with reply code 501 (frame error). Conformance test: amq_wlp_trace_01.
- If the recipient of a trace frame does not have a suitable trace handler, it MUST discard the trace frame without signalling any error or fault. Conformance test: amq_wlp_trace_02.

5.4.9 Heartbeat Frames

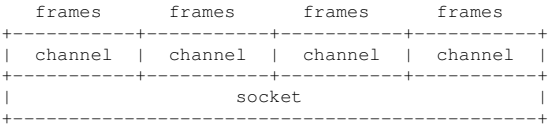
Heartbeat frames tell the recipient that the sender is still alive. The rate and timing of heartbeat frames is negotiated during connection tuning.

Guidelines for implementors:

- Heartbeat frames MUST have a channel number of zero. A peer that receives an invalid trace frame MUST raise a connection exception with reply code 501 (frame error). Conformance test: amq_wlp_heartbeat_01.
- If the peer does not support heartbeating it MUST discard the heartbeat frame without signalling any error or fault. Conformance test: amq_wlp_heartbeat_02.

5.5 Channel Multiplexing

We permit peers to create multiple independent threads of control. Each channel acts as a virtual connection that share a single socket:



A peer MAY support multiple channels. The maximum number of channels is defined at connection negotiation, and a peer MAY negotiate this down to 1.

Guidelines for implementors:

- Each peer SHOULD balance the traffic on all open channels in a fair fashion. This balancing can be done on a per-frame basis, or on the basis of amount of traffic per channel. A peer SHOULD NOT allow one very busy channel to starve the progress of a less busy channel.

5.6 Error Handling

5.6.1 Exceptions

Using the standard 'exception' programming model, we do not signal success, only failure. We define two exception levels:

1. Channel exceptions. These close the channel that caused the error. Channel exceptions are usually due to 'soft' errors that do not affect the rest of the application.
2. Connection exceptions. These close the socket connection and are usually due to 'hard' errors that indicate a programming fault, a bad configuration, or other case that needs intervention.

We document the assertions formally in the definition of each class and method.

5.6.2 Reply Codes

We use the IETF standard format for reply codes as described in IETF RFC 821. A reply code uses three digits, and the first digit provides the main feedback as to whether and how an operation completed. The second and third digits provide additional information. The reply codes can be processed by client applications without full knowledge of their meaning.

We use a standard 3-digit reply code. The first digit (the completion indicator) reports whether the request succeeded or not:

1 Ready to be performed, pending some confirmation.

2 Successful.

3 Ready to be performed, pending more information.

4 Failed, but may succeed later.

5 Failed, requires intervention.

6-9 Reserved for future use.

The second digit (the category indicator) provides more information on failures:

0 Error in syntax.

1 The reply provides general information.

2 Problem with session or connection.

3 Problem with security.

4 Problem with implementation.

5-9 Reserved for future use.

The third digit (the instance indicator) distinguishes among different situations with the same completion/category.

5.6.3 Channel Exception Reply Codes

When the server raises a channel exception it may use one of the following reply codes. These are all associated with failures that affect the current channel but not other channels in the same connection:

310=NOT_DELIVERED The client asked for a specific message that is no longer available. The message was delivered to another client, or was purged from the queue for some other reason.

311=CONTENT_TOO_LARGE The client attempted to transfer content larger than the server could accept at the present time. The client may retry at a later time.

403=ACCESS_REFUSED The client attempted to work with a server entity to which it has no access due to security settings.

404=NOT_FOUND The client attempted to work with a server entity that does not exist.

405=RESOURCE_LOCKED The client attempted to work with a server entity to which it has no access because another client is working with it.

5.6.4 Connection Exception Reply Codes

When the server raises a connection exception it may use one of the following reply codes. These are all associated with failures that preclude any futher activity on the connection:

320=CONNECTION_FORCED An operator intervened to close the connection for some reason. The client may retry at some later date.

402=INVALID_PATH The client tried to work with an unknown virtual host or cluster.

501=FRAME_ERROR The client sent a malformed frame that the server could not decode. This strongly implies a programming error in the client.

502=SYNTAX_ERROR The client sent a frame that contained illegal values for one or more fields. This strongly implies a programming error in the client.

503=COMMAND_INVALID The client sent an invalid sequence of frames, attempting to perform an operation that was considered invalid by the server. This usually implies a programming error in the client.

504=CHANNEL_ERROR The client attempted to work with a channel that had not been correctly opened. This most likely indicates a fault in the client layer.

506=RESOURCE_ERROR The server could not complete the method because it lacked sufficient resources. This may be due to the client creating too many of some type of entity.

530=NOT_ALLOWED The client tried to work with some entity in a manner that is prohibited by the server, due to security settings or by some other criteria.

540=NOT_IMPLEMENTED The client tried to use functionality that is not implemented in the server.

541=INTERNAL_ERROR The server could not complete the method because of an internal error. The server may require intervention by an operator in order to resume normal operations.

5.7 Security

5.7.1 Goals and Principles

We guard against buffer-overflow exploits by using length-specified buffers in all places. All externally-provided data can be verified against maximum allowed lengths whenever any data is read.

Invalid data can be handled unambiguously, by closing the channel or the connection.

5.7.2 Denial of Service Attacks

We handle errors by returning a reply code and then closing the channel or connection. This avoids ambiguous states after errors.

It should be assumed that exceptional conditions during connection negotiation stage are due to an hostile attempt to gain access to the server. The general response to any exceptional condition in the connection negotiation is to pause that connection (presumably a thread) for a period of several seconds and then to close the network connection. This includes syntax errors, over-sized data, and failed attempts to authenticate. The server should log all such exceptions and flag or block clients provoking multiple failures.