Concepts and Vision
A Background to the AMQ Project

version 0.9

iMatix Corporation <amq@imatix.com>

Revised: 2005/05/30

Contents

1 Introduction

1.1 Aim of this Document

This is a general introduction to AMQ, intended for a technically-minded readership. It explains the basic AMQ concepts, and goes into medium detail on all significant aspects of the architecture and technology.

Separate documents provide detailed guides to individual aspects of the architecture and technology.

1.2 Advanced Message Queues

AMQ ("Advanced Message Queues") is a broad and long-term project with the overall goal of creating commodity middleware for use in large-scale business and financial applications. The AMQ project has these specific short term goals:

1. To define an industry-standard wire-level protocol (AMQP/Fast).
2. To build a widely-usable reference implementation (OpenAMQ).

And these longer term goals:

1. To create a standard architecture for service-oriented networks.
2. To create a thriving open-source community (openamq.org).

1.3 Critical Success Factors

We have identified these three primary success factors:

1. Speed of the protocol and reference implementations.
2. Compatability with industry standards, especially JMS.
3. Availability of open source clients in many languages.

1.4 What is "Middleware"?

"Middleware" is a generic term for software that interconnects systems. We use the term specifically to mean software that passes messages between applications. The key characteristics of middleware, as we define the term, are:

- Application-level messages: the objects passed across the network are meaningful to client applications.
- Queueing and routing: the middleware must be able to queue messages internally, and route them to different clients in various ways.
- Asynchronous operation: messages are pushed through the network rather than pulled, with queues acting to buffer slower parts of the network.
- Selectable service levels: the client applications can explicitly choose between different combinations of speed and reliability.

An ideal middleware system must also be able to:

- Handle all kinds of data, opaquely, and without imposing any encoding or representation.

- Handle messages of any size up to multi-gigabytes.
- Handle both extremes of high-volume and high-reliability scenarios plus all sensible graduations in-between. In the first case throughput is critical but messages can be lost. In the latter case messages can never be lost, period.
- Provide several types of routing, including queues (where messages are distributed between consumers), pub/sub (where messages are published to as many subscribers as ask for them), topics (where messages are published according to a hierarchy of names, and content-based routing (where messages are routed according to key field values).

And further:

- To interoperate with or simulate other middleware systems.
- To run well on all boxes, from small clients to large servers.
- To be cheap enough to deploy without licensing concerns.
- To be easily adapted, extended, and if necessary, repaired.

And finally:

- To allow the creation of an abstract network of services.
- To provide ways for application developers to add services to the network.
- To provide ways for replicating data throughout such a network.

While there exist several large commercial middleware systems that provide all of the first set of requirements, there are no middleware systems - commercial or open - that also provide the second set. There are of course many other requirements for any kind of server.

The basic plan for making middleware from scratch is this: we start by making fast, reliable boxes that do the kinds of message passing we need. We then organise these boxes into more and more sophisticated networks by adding layers of services. To put it tritely: we build AMQ using AMQ.

1.5 Open Source Middleware

There is a fair amount of open source middleware. Mostly Java, mostly implementations of existing standard middleware APIs such as CORBA and JMS.

What is strikingly lacking in the open source middleware world is a robust answer to the problems actually facing enterprise software integration. There are no proposals for standard middleware protocols, there are no attempts to build commodity middleware servers such as the NCSA web server that became Apache. There are very few IETF working drafts and almost no attempt at standardisation except at the API level. CORBA seems to have been the last attempt at interoperability before the whole world moved to web services.

The main reason for the lack of classic open source / standards oriented developments in the middleware sector is probably the gulf between the academic and small open source teams, and the business world that builds systems large enough to need serious middleware. The small teams that might develop middleware are overwhelmed by Java, web services, and J2EE complexity. The larger businesses that sell middleware have had absolutely no incentive to break what is still a very lucrative market. And the client community is just starting to realise that they can, actually, solve this problem themselves.

AMQ is, to our knowledge, the first attempt to solve the middleware problem using standards from the wire up to the application. As a project, it is long overdue: middleware is a significant slice of software infrastructure that is still dominated by commercial vendor solutions and not moving in any serious way towards commodity open source.

1.6 Using Middleware in Real Life

1.6.1 Scales of Deployment

The scope of AMQ covers deployment at different levels of scale from the trivial to the mind-boggling:

1. Developer/casual use: 1 server, 1 user, 10 queues, 1 message per second.
2. Production application: 2 servers, 10-100 users, 10-50 queues, 30 messages per second (100K/hour).
3. Departmental mission critical application: 4 servers, 100-500 users, 50-100 queues, 60 messages per second (250K/hour).
4. Regional mission critical application: 16 servers, 500-2,000 users, 100-500 queues and topics, 250 messages per second (1M/hour).
5. Global mission critical application: 64 servers, 2K-10K users, 500-1000 queues and topics, 2,500 messages per second (10M/hour).
6. Market data (trading): 200 servers, 5K users, 10K topics, 100K messages per second (360M/hour).

As well as volume, the latency of message transfer can be highly important. For instance, market data becomes worthless very rapidly.

1.6.2 Extreme Scenario - Market Data

Market data is an extreme scenario at the upper-end of what we are aiming for with AMQ. We have set the benchmark at 100,000 events passing through a server (or server cluster) per second between a set of producer applications and a set of consumer applications. The occasional dropped message is acceptable, but dropout should be measurable.

To achieve this rate of messaging requires high-specification hardware, gigabit networking and significant tuning at all levels. Experience from existing projects shows that even OS context-switch time becomes significant at this rate of messaging. The topic space for event notification should be able to handle 10,000 topics with 50% of traffic volume going through 10% of the topics and delivered to 1,000 subscribers.

2 Challenges and Architectures

2.1 Designing a Successful Project

The AMQ project has had to face the same kinds of challenges that face all projects: to deliver high-quality results in a relatively short time and with a constrained budget.

Software projects tend to succeed or fail according to three main criteria:

1. The quality and relevant experience of the team.
2. The competence of the client.
3. The leverage and transparency of the toolkit.

In every failing software project (anything up to 75% of all projects by some estimates), one or more of these key criteria have failed. Most often, the people involved in a failing project are simply not very good. However, there are well-understood ways of handling mediocrity, if it is recognised. More often, perhaps very good people are forced to use unfamiliar or unstable technologies for political reasons (management saw a nice presentation).

It is rare to see projects fail for purely budgetary or planning reasons. The details of a project plan are far less important than ensuring that the team can ask questions freely, receive competent answers rapidly, and implement solutions with freedom of approach.

Tight deadlines and budgets can often improve the overall process by uncovering problems earlier. This is, perhaps why ”low-budget” projects can often be so much better than well-financed affairs. Not only in software - this applies to many other endeavours as well.

2.2 The General Solution

It’s a truism in software design that a general solution works better, is easier to maintain, and is often cheaper than a specific solution.

So it goes with AMQ - at every level we have tried to ignore the specifics of the problem to build general solutions. For example when designing the AMQ protocol we decided to treat messages as opaque binary MIME-typed blobs. This is more general than saying: ”messages will be XML documents that may encapsulate binary data”.

Another example of generality: the tools we built to allow us to build OpenAMQ are for the most part totally general and reusable tools.

2.3 Simplicity

Complexity is easy, but simplicity is truly valuable. The role of a software architect is largely about designing the most general structures that can be used in the most simple manner. The AMQ designs - for the protocol, server, and clients, have gone through numerous iterations so that we could remove unneeded concepts, turn specific cases into more general solutions, and in general attempt to do more with less.

The project is still complex. Middleware is not a trivial problem. But we have worked consistently to factor out all redundant functionality.

2.4 A Commodity Platform

A commodity product must use a commodity platform. That is, there is no benefit in developing a commodity product that has exotic dependencies. OpenAMQ must be able to build and run on a simple box with little more than a compatible compiler and standard libraries.

The platform we chose for our work was:

- ANSI C (mainly dictated by the need for performance and stability)
- A commodity OS: Linux, Solaris, or Win32 (dictated by our target users)
- TCP/IP (at least initially)

There are many useful libraries that fit into this mix: PCRE (Perl compatible regular expressions), zlib (the portable zip compression library), BDB (Berkeley DB), APR (Apache Portable Runtime), and we have added these into our project as needed.

The use of commodity technology is not the same as the use of commodity tools. We have made use of an extensive and sophisticated toolset, as explained in a separate section of this document.

2.5 Portability

Our target platforms are: Solaris, Linux, other POSIX boxes, and Win32. By "portability" we mean that the same code packages will build and work identically on all target platforms.

The iMatix approach to portability, defined in 1995 or so, is to put all non-portable code into a separate library that can be tuned for specific platforms. Application code then becomes 100% portable, with none of the conditional code that plagues most "portable" software.

The Apache project took a similar direction in 1999, building a portable runtime. Other projects such as Mozilla have done the same.

The OpenAMQ software uses both the Apache portable runtime, with some patches (this layer is not entirely mature) and iMatix portability libraries.

2.6 Protocol Design

A good discussion of protocol design can be found in RFC 3117, "On the Design of Application Protocols". The author of this RFC, Marshal T. Rose, contributed several IETF standards, including a protocol framework (BEEP, defined in RFC3080).

We did not use BEEP for AMQ, for several reasons: it uses XML wrapping that we felt was unnecessary and it does not have an actively-developed C implementation.

However, AMQP/Fast embodies many of the key design elements of BEEP. The main challenges are:

1. How to negotiate new sessions, including encryption, protocols and versions, capabilities, etc.
2. How to frame requests on a connection (defining how each request starts and ends).
3. How to allow many requests to work asynchronously in parallel (multiplexing).
4. How to allow many outstanding requests (pipelining).
5. How to report success/failure.
6. How to implement a functional model, i.e. at what "level" the protocol should operate.

We discuss the specific chosen solution to each of these in the "AMQ Protocol" section. In most cases our choice was driven by the desire to make the most general and high-performance solution possible.

2.7 Quality and Standards

Our first step in starting the project was to define a standard for defining standards, a document titled: "AMQ RFC001 - The OpenAMQ RFC system." The second step was "AMQ RFC002 - C Coding Standards".

2.7.1 Hand-Written Code

All programming needs to follow a strict style guide to remain readable. Our style guide for ANSI C has been developed over some time, and is remorselessly tuned for legibility. This is an extract from the RFC:

2.7.1.1 Spacing

- Code should be formatted to fit an 80 character width terminal [may push that to at most 90 char width].
- Indent with 4 spaces, never use tabs.
- Spaces outside () and [], not inside.
- Space after commas.
- No space after -> or . in accessing pointers/structures.

2.7.1.2 Naming Conventions

- All names are in English.
- Variable names are always lowercase with '_' as delimiter.
- Exported functions and variables are prefixed as lib_module_func.
- In rare cases, may abbreviate to lib_func.
- Macros are always capitalised.
- Type names end in _t,
- Functions or variables that are static to a single file are prefixed with s_,
- It is permissible to use short variable names within functions, as long as their definition is reasonably close to their place of use. (And functions should not be very long anyway.)
- Use somevar_nbr and somevar_ptr for local array indices and pointers in functions.
- Filenames are named lib_module.{c, h}.

2.7.2 Generated Code

GSL - the iMatix code generation language - is perhaps unique in that it is deliberately designed to produce highly readable code. There are some exceptions - for instance the code that implements the SMT state machines is unrolled to be very fast and thus becomes unreadable. The code that implements the iCL classes is sometimes poorly-indented.

But in general the code generation frameworks that we use so extensively produce code that a human programmer would be proud to write. Here is a random fragment of code generated for an OpenAMQ class (this code does topic publishing):

```
/*  -------------------------------------------------------------------------
    amq_vhost_publish
    Type: Component method
    Publishes a specified message to all interested subscribers in the
    virtual host.  Returns number of times message was published.  If the
    publish option is false, does not actually publish but only reports
    the number of subscribers.
    -------------------------------------------------------------------------
*/
int
    amq_vhost_publish (
    amq_vhost_t *  self,                /*  Reference to object              */
    char *   dest_name,                 /*  Topic destination name           */
    amq_smessage_t *  message,          /*  Message, if any                  */
    ipr_db_txn_t *  txn,                /*  Transaction, if any              */
    Bool     publish)                   /*  Actually publish message?        */
{
    amq_subscr_t
        *subscr;                        /*  Subscriber object                */
    amq_match_t
        *match;                         /*  Match item                       */
    int
        subscr_nbr;
    int
        rc = 0;                         /*  Return code                      */
    assert (self);
    /*  Lookup topic name in match table, if found publish to subscribers    */
    match = amq_match_search (self->match_topics, dest_name);
    if (match) {
        for (IPR_BITS_EACH (subscr_nbr, match->bits)) {
            subscr = (amq_subscr_t *) self->subscr_index->data [subscr_nbr];
            if (subscr->no_local == FALSE
            ||  subscr->client_id != message->handle->client_id) {
                if (publish)
                    amq_queue_publish (subscr->consumer->queue, message, txn);
                rc++;
            }
        }
    }
    return (rc);
}
```

Reading this code, you may wonder how on earth a code generator - surely a dumb thing - can produce something as intricate and specific as this function. The easy answer is "magic". The full answer is that the generated code is a layered mix of fully hand-written code, it is the mixing and layering techniques that supply the "magic".

2.7.3 Assertions

We make liberal use of assertions to make the code robust. Assertions also simplify error handling considerably. We do not remove assertions in production builds: they remain in the code permanently. Thus the example method shown above will assert that it is passed a valid object reference. If it ever gets an invalid reference, the server will abort.

2.7.4 Error Handling

Basically, we try to not handle internal errors. Wherever possible, errors in arguments, memory allocation, etc. are treated either as fatal, or as inconsequential. Our internal APIs return only success or failure (0 or non-zero), or in some cases a count or size.

When a particular layer detects an error it immediately reports the error to the console (so that it is captured in a log file) and then returns a 'failure' result. If the error is probably caused by a bug in the calling code, the program fails with an assertion.

Memory allocation failures are considered fatal. We do not attempt to recover from a failed malloc. We really do not want the server to start to swap on the way to running out of heap memory. Our strategy is to track the amount of memory used and reject new connections and/or messages when the memory is "full". This lets the system administrator set a resource limit for the server.

2.8 The Server Architecture

2.8.1 What Makes a Fast, Reliable Server?

Apart from the obvious requirements of writing good code that is reasonably efficient, the key to building a fast server is to reduce the cost of servicing individual connections, and the key to building a reliable server is to use finite state machines (FSMs) that operate and communicate asynchronously.

2.8.2 Server Connection Handling

When iMatix began designing server toolkits in 1995, the classic model was "one connection, one process", possibly with process pooling. This model breaks-down rapidly, with a limit of 30-100 connections per server (at which point system memory gets full).

We designed a single-process architecture (SMT) which uses pseudo-threads (internally managed by the architecture, with no help from the operating system) to eliminate process duplication. A server built on this model (e.g. our Xitami web server) can handle hundreds or thousands of connections with no serious impact on system memory.

When we reach several hundred connections, a new problem arises, namely the use of system calls like "select" that involve linear searching. We can use alternate system calls like "poll" that are more efficient, and our current version of SMT does this. Servers built on this design are rapid even with very many connections.

On larger boxes, however, a single-process server shows another weakness: it cannot exploit multiple CPUs. Even the fastest single-process server will be out-performed by a less efficient server that can run on 4 or 8 CPUs.

Modern software uses multiple system threads to exploit multiple CPUs. Operating systems like Solaris and recent Linux kernels are extremely good at scheduling threads and switching between them.

Thus, a modern server must use multiple system threads. We are (as this text is being written) modifying SMT to use a separate system thread per pseudo-thread.

Finally, to handle very large numbers of connections (the so-called "C10K problem"), we have to move to a more radical model still: a small number of OS threads handling multiple connections, and a fully asynchronous I/O model. A portable solution for this is somewhat beyond the state of the art (e.g. the Linux 2.6 kernel supports asynch. I/O but not for sockets), but we are developing a solution.

The predecessor to SMT, a pseudo multi-threading framework for OpenVMS written in 1991 (and still used today) was entirely based on asynchronous I/O (using OpenVMS system traps) and could handle 500-1000 concurrent interactive users on modest hardware (e.g. an AlphaServer DS10, 675Mhz 256MB RAM).

2.8.3 Finite State Machines

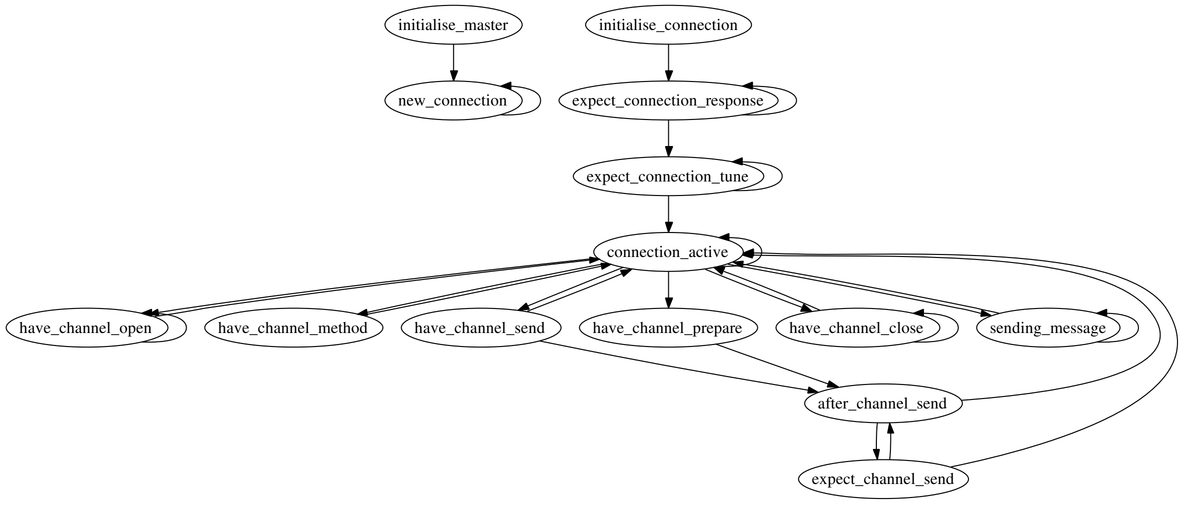
Underlying all these different I/O and threading models is a finite state machine architecture. FSMs are particularly good for servers because we can make them fully stable. That is, a well-designed FSM handles errors so explicitly that the server never falls into the type of ambiguities that generally cause failure.

Our FSM model is based on work done in several software engineering tools (ETK, Libero) since 1985. The model is simple and generic, based on this elementary design unit:

```
state
    event -> next-state
        action
```

Where a state consists of 1 or more events, each which take the FSM to a new 'next state', after executing zero or more actions. We add concepts such as state and event inheritance, exception events, and called states.

The core of the server consists of this state machine:



The source code of the state machine is an XML 'program'. Here is a fragment of this code:

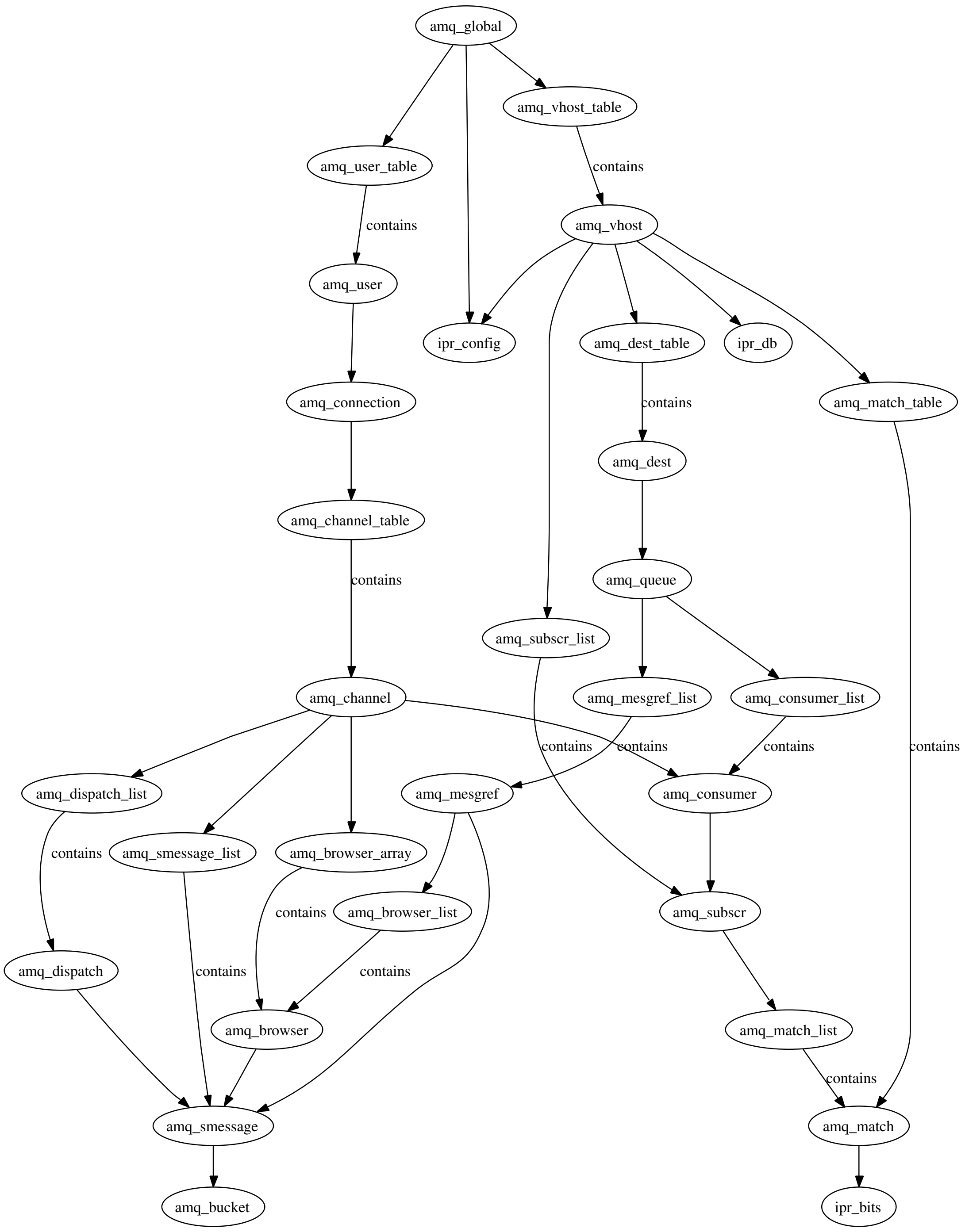
```
<state name = "initialise connection">
    <event name = "ok" nextstate = "expect connection response">
        <action name = "read protocol header" />
        <action name = "check protocol header" />
        <action name = "send connection challenge" />
        <action name = "read next method" />
    </event>
    <event name = "connection error" nextstate = "">
        <action name = "close the connection" />
    </event>
</state>
<action name = "read protocol header">
    s_sock_read (thread, tcb->frame_header, 2);
</action>
```

Note the ANSI C code wrapped inside XML tags. We use this technique extensively in our code, as we describe later.

The server finite-state machine is explained in more detail in the "OpenAMQ Server Architecture" document.

2.8.4 Server Classes

The server is built as a hierarchy of classes. The following figure shows the way classes use each other. We show two types of usage, classes that hold a single instance of another class, and classes that act as containers for a set of child classes:



We use the iCL framework to write the classes. This framework is described in more detail later. Its advantage for our work is that we can maintain a very formal organisation of code. The fact that iCL generates huge amounts of perfect C code for free is useful too.

The server classes are explained in more detail in the "OpenAMQ Server Architecture" document.

2.8.5 The Matching Engine

The key to the performance of the matching is the use of "inverted bitmaps", in which search tables are pre-calculated and stored as result bitmaps. The matching engine relies on these specific components:

1. A parsing module that extracts search terms from a subscription or from a message header.
2. A bitmap module that provides read/write/search and logical operations on large bitmaps (16k bits).
3. A hash table that holds search terms and their corresponding bitmaps.

The basic algorithm is broken into two phases - subscription indexation and message matching. Indexation works as follows:

1. Subscriptions are numbered from 0 upwards. The numbering scheme re-uses numbers when subscriptions are cancelled. This guarantees an upper limit on the subscription number.
2. On a new subscription we parse the key fields on which the subscription acts into a set of search terms.
3. For each search term, we load the corresponding bitmap and set the bit corresponding to the subscription. We save the modified bitmap.

Message matching works as follows:

1. We parse the message header into a set of search terms.
2. For each search term we load the corresponding bitmap, if any.
3. We logically AND or OR the bitmap with a rolling result bitmap, depending on how the subscription was constructed.
4. After processing all search terms for the message we are left with a bitmap that indicates the subscriptions that should receive this message.

In tests, the basic matching algorithm can perform about 2 million matches per second real-time on a 1Ghz CPU.

2.9 Client Architecture

2.9.1 General Client Architecture

The OpenAMQ clients form a large part of the whole project. In theory we could make a full from-the-wire implementation for each programming language - C, C++, Java, Perl, .NET, COM.

What we have chosen to do is to define a general architecture that can be implemented in all clients, and then to standardise and generate as much of this architecture as we can.

Our goal is that all OpenAMQ clients, whatever the language, use a similar set of concepts and tools, and provide the application programmer with a similar API. This is not yet entirely the case.

2.9.1.1 The Framing Layer

AMQP/Fast defines a set of methods that map onto frames (each method is a specific frame). The frames are formally defined in the amq_frames.xml document. For example:

```
<frame name = "connection challenge">
    <field type = "octet"    name = "type"      value = "10" />
    <field type = "octet"    name = "version"   >Protocol version</field>
    <field type = "shortstr" name = "mechanisms">Security mechanisms</field>
    <field type = "table"    name = "challenges">Challenge fields</field>
</frame>
```

//TODO: amq_frames.xml will be replaced with an IDLRich definition.

Using GSL code generation, it is a fairly simple matter to turn the formal frame definitions into marshalling code. We have done this in C, Perl, Java. This is a snippet of the generated C marshalling code:

```
if (IS_Connection_CHALLENGE (source)) {
    frame->type = FRAME_TYPE_Connection_CHALLENGE;
    frame->body.connection_challenge.type = *source++;
    frame->body.connection_challenge.version = *source++;
    string_size = *source++;
    memcpy (frame->body.connection_challenge.mechanisms, source, string_size);
    frame->body.connection_challenge.mechanisms [string_size] = 0;
    source += string_size;
    GET_SHORT (string_size, source);
    frame->body.connection_challenge.challenges =
        ipr_longstr_new (source, string_size);
    source += string_size;
    frame->size = source - buffer;
}
```

The framing layer is entirely generated. The same XML specifications are used in the server and in the clients, and to produce the AMQP/Fast RFC specifications. As well as eliminating a lot of redundant work, this ensures that frame fields are consistently named in all our AMQP/Fast implementations.

2.9.1.2 The Transport Layer

The transport layer handles socket connections to the server, and reads and writes raw frames.

This layer is specific to each language, and even in a single language there may be several implementations. For instance in C we have an SMT implementation and a 'classic' implementation using OS threads.

2.9.1.3 The Protocol Layer

The protocol layer consists of one or more FSMs, defining the connection and channel states, and the validity of frames in each state. It must also handle API requests ('methods') and their validity in each state.

We are working to define a single set of "categorical" FSMs for the protocol layer, in the same way as we have a single specification for frames. At present the FSMs are built by hand using different tools. GSL lets one write an ad-hoc FSM generator in a few hours, so the real cost is not the burden of maintaining extra tools, but of maintaining multiple instances of what should be the same FSMs.

2.9.1.4 The External APIs

The Protocol Layer exports one or more APIs to the calling applications. These APIs may themselves be wrapped to provide something more familiar to naive developers.

AMQ aims to be a standard and we also prefer to use existing standards where possible. In the middleware domain, the dominant standard is JMS, the Java Messaging System, defined by Sun and large middleware vendors.

JMS has several strong advantages:

- It is widely used and well-known, so supporting JMS means we gain instant credibility and entry into an existing skill pool.
- JMS implements most mature middleware models. Basically, if we can "do JMS", we satisfy a large portion of the market.

JMS has two disadvantages:

- It is in principle licensed only for Java use. We expect that it is possible to build a JMS-like API without falling foul of these license terms but we are not 100% certain at this stage.
- JMS was designed with weaknesses. This was, we assume, a deliberate policy to maintain a gap between JMS providers and "real" middleware products. All serious JMS implementations make their own extensions. JMS is not standard unless the developer stays away from these extensions.

Our goal is to support a JMS-like API. In Java, this should look as close to JMS as the JMS license and copyrights will allow. In other languages, we will use the same names and class hierarchies.

AMQP/Fast provides other operational models - file transfer, streaming - that do not fit the JMS API, and we will provide specific APIs for these.

At the time of writing, we are still working on these APIs. The existing clients do not implement JMS but somewhat simpler, cleaner APIs.

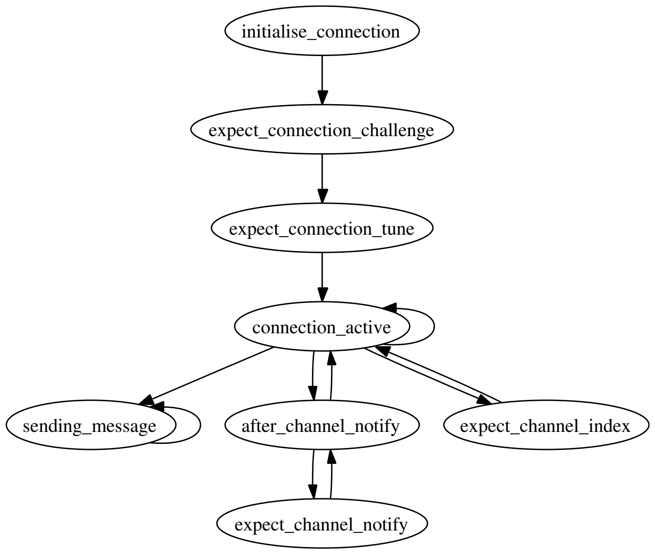
2.9.2 OpenAMQ Client Implementations

There are several different client architectures:

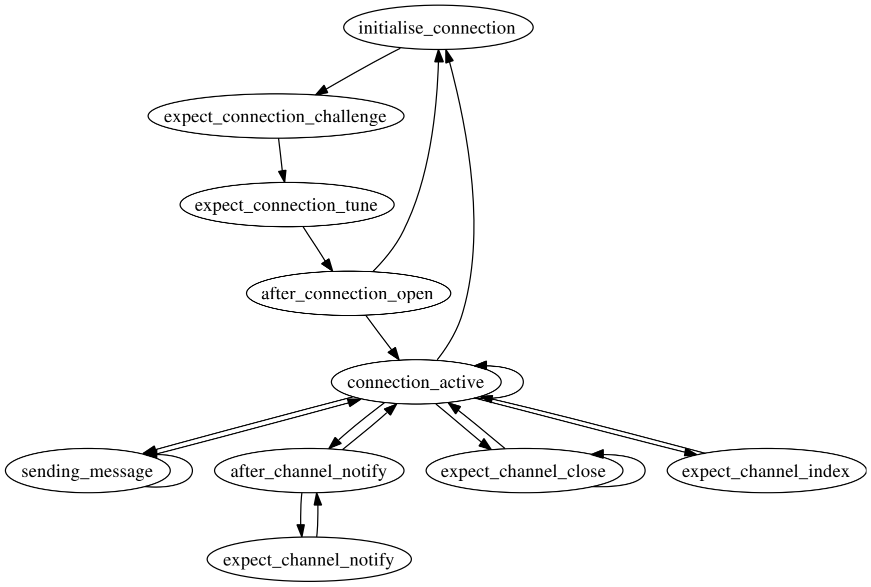
- A full general-use Java client. This provides an API compatible with AMQ RFC014/0.1. This client is built using a generated framing layer, FSMs for the protocol layer, and it uses Java streams for the transport layer.
- A full general-use ANSI C client. This provides an API compatible with AMQ RFC014/0.1. This client is built using a generated framing layer, FSMs for the protocol layer, and it uses OS multithreading over sockets for the transport layer.
- A COM client. This client provides an API compatible with AMQ RFC015/0.4. It uses the synchronous SMT-based client (described below).
- An asynchronous kernel client in C/SMT. (The "kernel" is the core library used to construct OpenAMQ.) This uses the same iMatix SMT threading technology as, and is designed to be integrated into, the OpenAMQ server.
- A synchronous kernel client in C. This uses the OpenAMQ server technology but is intended for general use.

The full general-use clients are based on similar architectures and intended for use in standard single or multi-threaded applications. The C/SMT client is specifically for C/SMT applications, which includes the OpenAMQ server itself but could also mean other servers we build using C/SMT.

This are some FSM diagrams for the clients. This is the asynchronous kernel client:



This is the synchronous kernel client:



2.10 An Open Source Project

2.10.1 Goals and Economics

AMQ was conceived from the start as an open source product. This vision is probably the most radical and defining aspect of the entire project.

The AMQ project, funded by a large institution and built by a specialist engineering firm, represents a new way of building software systems. As it grows, AMQ should draw support from many other institutions and firms. To a large middleware consuming firm, even a significant investment in such a project is much cheaper than the tangible cost of commercial alternatives and the intangible costs of working with arbitrary, incompatible, and closed systems.

The economics of the AMQ project are simple: every Dollar spent on creating this product will generate ten, a hundred Dollars of value as the cost of interconnecting applications comes down to zero, allowing larger and more efficient software architectures.

Using proprietary infrastructure is less advantageous: the commercial interest is invariably opposed to the customer's interest.

We believe that in a decade, this model of collaboration between large software consumers and smaller open source producers will be commonplace, even conventional. Infrastructure is expensive to make, and open source is a way - perhaps the only long-term way - to spread these costs effectively.

But in today's world, the sponsorship of AMQ by JPMorgan Chase is a bold innovation.

2.10.2 How Open Source Works

Most people by now understand what open source (aka free software) is, at least superficially - communities of dedicated programmers who for some alien reason decide there is more fun, interest, or ironically, profit to be had in giving their hard-written software away to all and sundry, rather than doing the right and natural thing and charging people money to use it.

What few people understand is how this process works. Since we at iMatix have been writing open source as an essential part of our business activities for almost 15 years, it is safe to say that we have a certain understanding, or at least opinion backed up by long experience, on this matter.

Open source, or free software, is a phenomenon that has arisen spontaneously as the cost of communications has fallen over the last decades. There were no government initiatives, no great academic sponsors. (The first free software philosophers were rebelling against the academic tradition of turning student and PhD work into commercial spin-offs.) One can almost say that as communications get cheaper, programmers will tend to collaborate, building the tools they need, then building infrastructure, then entire systems.

The attitude of commercial software producers towards this phenomenon is mixed: some see open source as a fashion, a threat, or a left-wing conspiracy. Others see it as a rather perplexing opportunity. It seems clear that open source (or rather, the commoditization process that it represents) splits the industry into two distinct halves: those who see the future as a sea of cheap software supporting flotillas of services, and those who make their money from selling expensive water.

While open source seems a somewhat chaotic, and definitely anarchic process, and while asking ten open source developers "why they do it" will result in twelve different answers, the mechanisms of the process are quite simple.

First, individual programmers always look to work on projects that will maximise their opportunity for growth and new opportunities. In a technical career, success comes from being an expert (even an expert generalist). The best programmers seek the hardest challenges.

Second, since the IT world is vast, and complex, and mostly filled with junk (following Sturgeon's Law), smart programmers look for other smart programmers with whom to work, and (importantly) against whom to compete. It is not possible to measure one's success in a vacuum, nor in a crowd.

Third, since keen programmers are involved in a creative intellectual process, rather than a commercial process, the easiest way to demonstrate one's skill is to publish one's work. The invention of open source licenses - primarily the GPL - guarantees a fairness of use that satisfies the instinct for fair play.

Fourth, the cheaper that communications become, the easier it is for these smart programmers to find each other, work together on small projects, and collaborate over long periods of time on larger projects. Cheap communications bring people together.

Fifth, when a group of elite programmers get together, they start to look for "interesting" challenges to work on. Long before open source became an obvious trend, there were many very successful collaborative projects. Most of these filled niches: fractint, the Scandinavian "demo" culture, emacs. In each case competition with other projects was a key driver.

The Internet boom that started in the mid-1990's pushed this process exponentially.

So open source developers are mainly an elite - the very best programmers - who enjoy working on the cutting edges of IT, where there is an active R&D process, risk, and potentially huge reward. They compete to build the most elegant, and the most useful, structures.

Open source / free software (as compared to other types of collaborative project) is particularly effective because every improvement is added back into the process. There is no proof as yet, but one can argue that the GPL license creates better software, faster, than the BSD license, because of this effect. We'll come back to this argument.

2.10.3 Open Source as an Economy

A common misconception about open source is that because it is "free" it is somehow a charity operation where programmers work bene-vola because they want "to contribute".

This is, however, wrong. When Adam Smith said: "It is not from the benevolence of the butcher, the brewer, or the baker, that we expect our dinner, but from their regard to their own interest", he was accurately describing a world in which self-interest creates mutually-beneficial structures.

Open source contributors are attracted for different reasons, depending on how far they understand and identify with the technology at hand. We can identify the self-interest of each role, while seeing that the overall structure serves everyone:

- "Users" will evangelise (seeking security in the company of others using the same technology).
- "Power users" will help others who have problems (seeking the kudos that comes from helping others).
- "Pundits" will discuss the technology in public forums (seeking the fame that comes from being able to accurately identify trends and future winners).
- "Insiders" will take on parts of the testing process (seeking better familiarity with a technology that may become an important part of their skill set).
- "Players" will delve into the technology itself, taking on smaller roles in the process (seeking the kudos and fame that can come from being on a winning team).
- "Key players" will take on major roles in the project (seeking to impose their ideas, turn a small project into a major success, or otherwise earn a global reputation).
- "Patrons" will provide financial support to the project (looking to sell services, often to the users, that require the technology to succeed and be widely used).

The naive view of open source focuses only on the players, ignoring the wider economy of interests. A successful open source project must attract and support all these classes of people (and others, such as the "troll", who vocally attacks the project in public forums, thus stiffening the resolve of the users and pundits who defend it).

Thus we can understand the needs of each role:

- Users need a pleasant and impressive product so they can feel proud about showing it to others.
- Power users need forums and mailing lists where they can answer questions.

- Pundits need pre-packaged press releases, insider tips, and the occasional free lunch. Some controversy also helps.
- Insiders need regular releases, frequent improvements, and forums where they can propose ideas for the project.
- Players need extension frameworks where they can write their (often sub-standard) code without affecting the primary project.
- Key players need badges of membership, and access to the right tools and support.
- Patrons need a high-quality and stable product that supports their services and additional products.

The only people working full time, and usually professionally, on an open source project are the key players. All the others will take part in the project as a side-effect of their on-going work or hobbies.

While a traditional software company must pay everyone in this economy except the users, an open source economy must only pay the key players, who make up perhaps 2-5% of the total. Further, the key players will work for significantly less than the market rate, since they also derive a real benefit from working on successful projects, which I call the open source "payload". The most important part of a future programmer's CV is the section titled "Open Source Projects". This is the payload. It translates directly into dollars, proportional to the impact and importance of the open source projects involved.

When compensation plus payload does not cover the cost of working on a project (in terms of loss of compensation for alternative work), the key player will suffer "burnout" after 12-18 months, more or less depending on the person's tenacity.

2.10.4 Key Requirements

We can summarise the above discussion into a set of key requirements for a successful open source project:

1. Quality at all levels. The product must look and feel like a perfect machine: our target users identify with technology, not cosmetics.
2. An arena for discussions, at least email lists, and probably also interactive forums.
3. A clean web site with downloadable presentations, press releases, documents, flyers, and other material.
4. A regular release schedule with raw, fresh, and well-cooked packages (i.e. satisfying pioneers, early adopters, and late adopters).
5. An extensible software framework which provides players with a way to write addons of all types without wasting the time of the key players.
6. A forum or distribution channel for players to exchange and discuss their contributions.
7. A project identity and personality that attracts key players and patrons.
8. A project financial structure protects the key players from burn-out.
9. A clear financial rationale for the project that attracts patrons and ensures they get value for money.
10. Some arbitrary and controversial technical and/or licensing choices that keep the trolls happy and ensure the project gets discussed often on popular forums like Slashdot.

Insofar as we can "market" an open source project like OpenAMQ into success, the categories I've defined are the market segments that need to be addressed. For instance, a project's web site should viewed from the perspective of each of these roles and the question asked, "does this make sense to me?"

2.10.5 Open Source Licenses

There are numerous open source licenses, but modern projects tend to use one of two main alternatives. On the one hand there is the BSD model, also used by Apache, which says "this code is free to use and share but you must respect our copyright notices". On the other hand there is the GPL model, the basis of Linux, which says "this code is free to use and share, and any modifications you distribute as binary must also be shared as source."

The two approaches are strongly influenced by philosophies. The BSDL originates from an academic desire to spread knowledge widely. The GPL originates from the viewpoint that software-as-property is somehow wrong. It even promotes the notion of "copyleft" as an alternative to copyright, though the license is only enforceable through copyright law.

Many academics and most commercial interests dislike the GPL because it "infects" projects. Even iMatix, who distribute their own projects as free software under the GPL, prefer to work with BSDL libraries. Why? Because we can relicense our GPL code for commercial clients, and provide this together with BSDL code. This would be impossible if we relied on GPL code.

From the viewpoint of the success of open source projects, there is contradictory evidence. On the one hand, BSDL projects like Apache have held a market lead against GPL alternatives. On the other hand, Linux has beaten the originally superior OpenBSD and FreeBSD systems into obscurity.

It is hard to generalise about the kinds of project that each license attracts but anecdotal evidence suggests that BSDL projects for the most part are more conservative than GPL projects. That is, over time they have clearer goals, more consistent structure and quality, and less chaos. GPL projects tend to be messier but more lively. Perhaps the GPL attracts more extreme mindsets, or the "contribute everything" approach means more people necessarily get involved. Also, GPL projects can absorb BSDL source code, but not vice-versa. Whether the choice of license is cause or effect, it does appear that the GPL coincides with a more dynamic, risk tolerant, and overall more productive open source economy.

For AMQ, we'd prefer to use the GPL as it fits our view of open source as an economy. The argument is that the GPL, though harder to "sell" to our users, would create a more viable long-term economy around the AMQ technologies. However, we'd recommend using the BSDL (rather, the Apache License) for practical reasons: it is what our target users will want.

2.11 Escaping the "Version 2 Syndrome"

A project like AMQ solves a problem that is very well defined - after twenty or more years of work, middleware can basically be reduced to a fixed set of queueing and routing mechanisms with a large amount of wrapping - and for which there is a huge market. In a large global enterprise, there are dozens, perhaps hundreds of potential applications where commercial or ad-hoc middleware can be placed aside in favour of a simpler standard solution.

The user community is sophisticated, having worked with middleware of all flavours, and able to rapidly define the most ambitious goals in terms of performance, stability, and functionality.

On the one hand this maturity is essential if we are to avoid making too many mistakes while discovering the ideal solutions to the many problems we have to solve.

On the other hand, this maturity of vision can become a danger to the banal realities of building the project. Too many features, added too soon, turn the best source code into mush.

We describe our process for filtering functionality while keeping the door open to future extensions.

2.11.1 Packing Light

The first rule we apply to any given functionality is: do we need it now? There are several criteria for this:

- Do our first target applications need it?
- Does it add essential credibility to the AMQ project?
- Do we know how to make it?
- Can we make it reasonably quickly and cheaply?

If the answer is 'yes' to most or all of these questions, we implement the functionality. If 'no', we put it aside.

Packing light means making an unencumbered design. Every unused or idle element of a design adds weight and cost. Knowing what not to do (yet) is as important as knowing what to do.

2.11.2 Extensibility, Not Functionality

The key to being able to say "we will do it later" is that the work will be as easy and fast to do (possibly more so) in the future than now.

We aim to keep all structures general so that they can be extended and added to easily. For example, we know that the AMQP/Fast specifications are not complete, that it is missing large and (in some scenarios) vital chunks, but we also have a good idea of where those chunks will go and we do what we can to minimise the effect those future changes will have on the design.

2.11.3 Change Management

We treat change as an essential part of the design process. That is, change in requirements, in design decisions, in the code, everywhere. Every iteration of the software defines the design of the next.

This is possible because we rely heavily on code generation to leverage our programming. It is simply a matter of inertia: fewer lines of code to refactor means less work and fewer mistakes.

2.11.4 What Ships in Version 1.0?

The main functionality planned for version 1.0 of OpenAMQ is:

- Queues and topics.
- Persistent and non-persistent messages.
- Routing by queue name, topic hierarchy, message header fields.
- Transaction management.
- Main content domains: JMS, file transfer, streaming.
- Virtual hosts.
- Client APIs for C, C++, Java, COM, Perl.
- Project support structures: web site et al.
- SASL security.
- Access-control lists.
- Simplified selectors.

Functionality that will probably not be included in version 1.0:

- XA transactions.
- Full JMS selectors.

And functionality that we still have to look at in more detail:

- System queues for remote administration.
- Web-based administration interface.
- Clustering and replication.
- Dynamic cluster organisation facilities.
- Server-side applications.
- Server plugins and filters.

2.12 Prior Art

2.12.1 The OpenAMQ Matching Engine

The OpenAMQ matching engine is based on work done in 1980-81 by Leif Svalgaard and Bo Tveden, who built the Directory Assistance System for the New York Telephone company. The system consisted of 20 networked computers serving 2000 terminals, handling more than 2 million lookups per day. In 1982 Svalgaard and Tveden adapted the system for use in the Pentagon (Defense Telephone Service). This system is still in operation.

2.12.2 The SMT Multithreaded Kernel

The SMT multithreaded kernel is based on work done in 1993 as part of the ETK Toolkit implementation for OpenVMS/ACMS. This work formed the basis of a transaction-processing front-end used by Delight Information Systems, processing several million travel bookings per year.

2.12.3 Code Generation Technology

iMatix code generation technology is based on research and development done by Leif Svalgaard and Pieter Hintjens from 1985 to the present day in the ETK toolkit (Sema Group Belgium, 1985-1995), and by Pieter Hintjens and Jonathan Schultz from 1995 to the present day in these iMatix products: Libero, htmlpp, GSL, iAF, Aqua, UltraSource, XNF, iCL, and SMT.

3 The AMQP/Fast Protocol

This section is the first half of the AMQ RFC 006, which defines the AMQ Protocol. The second half of this RFC is a full description of each method. This is still under preparation. When the RFC 006 is complete, this section will be edited into a shorter concepts guide.

3.1 Discussion

3.1.1 Overview

3.1.1.1 Purpose and Goals of AMQP/Fast

AMQP/Fast is designed to be the high-performance member of the AMQP family of standardised general-purpose message-oriented middleware wire-level protocols.

A "wire-level protocol" can be a simple as a "this is how we define packets on the wire". AMQP/Fast is more ambitious than that. It tries to provide formal answers to a series of questions that we must answer in order to implement useful messaging between protocols.

We want to be able to:

- Construct logical "messages" out of data of any type and size, and move these message efficiently across a network of clients and servers.
- Provide queueing and routing semantics for messages so that the behaviour of the server is unambiguously defined by the protocol.
- Define explicit operational semantics for common messaging models, including JMS, file transfer, and streaming, at the protocol level.
- Define quality-of-service levels semantics so client applications can explicitly choose appropriate trade-offs between speed and reliability.
- Create a fully-asynchronous network so that messages can be queued when clients are disconnected, and forwarded when they are connected.
- Allow interoperation and bridging with other middleware systems.
- Allow implementation in any language, on any kind of hardware.
- Avoid license and patent concerns that might hinder adoption of the protocol as a standard.
- Allow the marshalling layers of protocol implementations to be easily generated using code generation technology.
- Allow the creation of an abstract network where services, and their data, can move around the network opaquely to the clients.

3.1.1.2 Intended Operating Lifespan

AMQP/Fast is designed to give a useful lifespan of 50 years or more. Our goal is that an AMQP/Fast peer will be able to operate continuously with no upgrades or incompatibility for at least this duration, without requiring "legacy support". One should be able to build a client or server into physical infrastructure.

While the protocol version may and will change, the protocol mechanics (framing, method structures, etc.) must operate unchanged for the full intended lifespan of the protocol, allowing full and perfect forwards compatability with all future versions of the protocol.

We have applied "Moore's Law" - the theory of exponential growth of capacity of technology - to all capacity limits to identify and eliminate potential future bottlenecks, specifically for:

1. Message sizes: the largest messages (files) are today around 20GB. We expect this to grow by 50% per year, reaching the limit defined by a 64-bit size in 50 years.
2. Frame sizes: IPv4 is limited to 64KB frames, IPv6 to 4GB frames. Ethernet itself is limited to 12000 byte frames due to its CRC algorithm. We expect the maximum networking frame size to grow by 50% per year, in large leaps. We will thus reach the limit of 64-bit sized packets in 50 years.
3. Sequencing: we use method sequencing to allow matching of replies and errors with requests, so called "synchpoints". The method sequence number is 32 bits. This allows 4G methods to be exchanged between synchpoints. We do not see this as a limitation in any scenario, present or future.
4. Protocol classes and methods: the current protocol defines about ten classes and about ten or fewer methods per class. We expect that new versions of the protocol will add classes and methods at a constant rate of 1 to 5 per year. The limit of 255 classes and methods per class should be sufficient to last until 2055.
5. Channels: the limit of 4G channels allows growth of 50% per year from an estimated usage of 10 channels per connection today.
6. Timestamps: we use 64-bit time-stamp values.

3.1.1.3 An Extensible Functionality

AMQP/Fast is intended to be extensible in several directions, including new directions totally outside the scope of the protocol as it is designed today. These are the aspects of AMQP/Fast that have been deliberately designed to be extensible (in order of increasing generality and power):

1. Adding new properties to content domains.
2. Adding new arguments to methods.
3. Adding new methods to classes.
4. Adding new content domains.
5. Adding new classes.
6. Adding new frame types.
7. Adding new protocols.

All of these should be feasible while maintaining full backwards compatability with existing implementations.

3.1.1.4 Ease of Implementation

Our goal is to achieve some of the cost-benefit ratio of protocols such as SMTP and HTTP, where a simple client can be trivial to build but a full client can be very sophisticated. Keeping AMQP/Fast accessible to simple clients is possible if we hold to these design rules:

1. The use of all complex functionality (e.g. more sophisticated data types or structures) must be optional.
2. The protocol must be able to operate entirely synchronously, since an asynchronous model - though efficient and reliable - is a barrier for simple implementations.
3. The protocol must be formally defined so that significant parts of a client or server protocol interface can be mechanically generated.

3.1.1.5 Guidelines for Implementors

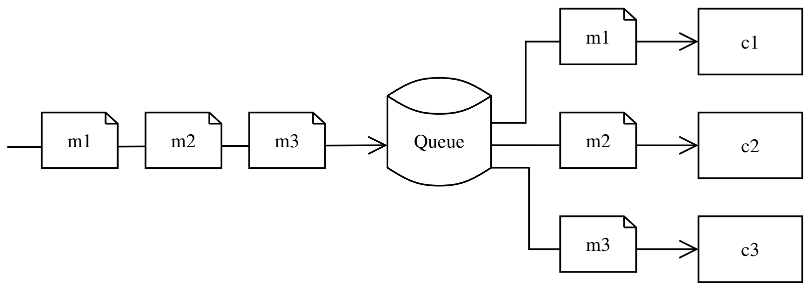
- We use the terms MUST, MUST NOT, SHOULD, SHOULD NOT, and MAY as defined by IETF RFC 2119. Where possible we note the security implications of the guidelines.
- We use the term "the server" when discussing the specific behaviour required of a conforming AMQP/Fast server.
- We use the term "the client" when discussing the specific behaviour required of a conforming AMQP/Fast client.
- We use the term "the peer" to mean "the server or the client".
- All numeric values are decimal unless otherwise indicated.
- Protocol constants are shown as uppercase names. AMQP/Fast implementations SHOULD use these names when defining and using constants in source code and documentation.
- Property names, method arguments, and frame fields are shown as lowercase names. AMQP/Fast implementations SHOULD use these names in source code and documentation.
- Protocol methods are defined using a formal language as defined in AMQ RFC xxx.

3.1.1.6 AMQP Standard Models

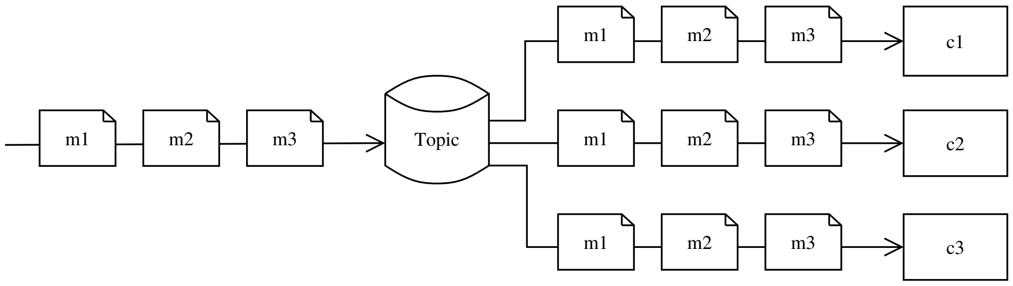
All AMQP protocols including AMQP/Fast implement a common and interoperable queueing and routing model based on these concepts:

1. Application-level messages ("content domains").
2. Persistent and non-persistent message storage ("destination").
3. A distributor mechanism ("queue").
4. A multiplier mechanism ("topic").
5. A way for clients to send messages to destinations ("publish").
6. A way for clients to request messages from destinations ("subscribe").
7. A way for clients to request individual messages from destinations ("browse").
8. A way for clients to inform the server when a message has been successfully processed ("acknowledgement").

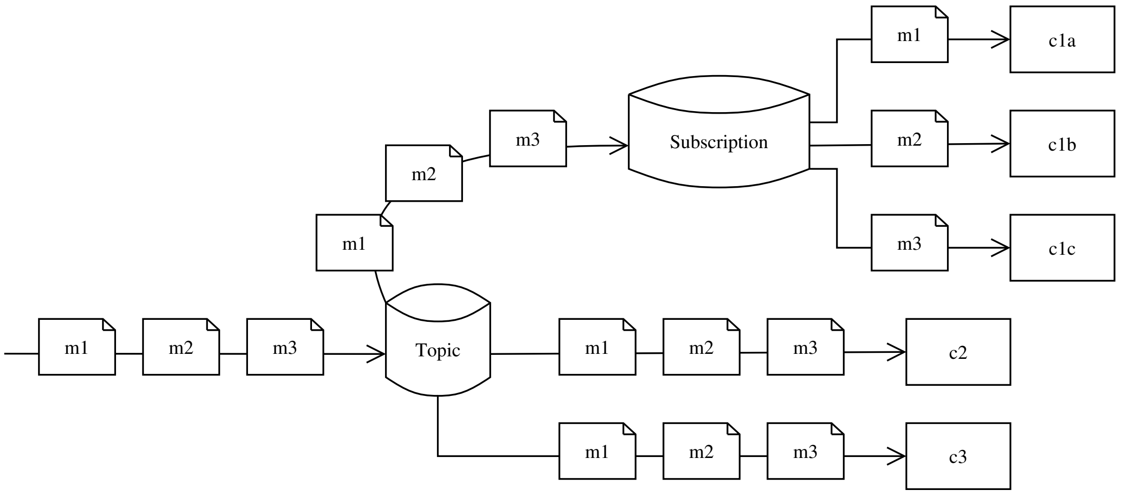
The AMQP "queue" concept acts to distribute messages ("m1", "m2", "m3") between a set of consumers ("c1", "c2", "c3"):



The AMQP "topic" concept acts to multiply messages to a set of consumers:



The AMQP "subscription" concept acts like a queue:



3.1.1.7 Definitions

Connection A network connection, e.g. a TCP/IP socket connection.

Channel A bi-directional stream of communications between two AMQP/Fast peers. Channels are multiplexed so that a single network connection can carry multiple channels.

Client The initiator of an AMQP/Fast connection or channel. AMQP/Fast is not symmetrical. Clients produce and consume messages while servers queues and route messages.

Server The party that accepts client connections and implements the standard AMQ queueing and routing functions.

Peer Either party in an AMQP/Fast connection. An AMQP/Fast connection involves exactly two peers (one is the client, one is the server).

Frame A formally-defined package of connection data. Frames are always written and read contiguously - as a single unit - on the connection.

Extended_Frame A frame capable of using more channels and carrying more data than a normal frame.

Meta_Frame A specific type of frame that has the sole purpose of indicating that the next frame will be an extended frame.

Protocol_Class A collection of methods that deal with a specific type of functionality.

Method A specific type of frame that passes instructions from one peer to the other.

Content Application data passed from client to server and from server to client. AMQP/Fast content can be structured into multiple parts.

Header A specific type of frame that describes content.

Body A specific type of frame that contains raw application data. Body frames are entirely opaque - the server does not examine or modify these in any way.

Trace_Frame A specific type of frame that carries information to a "trace handler", an abstract processing unit that may be embedded in an peer.

Message A specific type of content, the application-to-application message.

Destination A queue, topic, or subscription, able to hold and route messages of various types. A server manages access to a set of destinations.

Static_Destination Destinations that are configured by the server administrator.

Dynamic_Destination Destinations that are created and managed by clients via protocol methods. Unlike "temporary" destinations, dynamic destinations provide true disconnected asynchronous processing in which a client can create a response queue, then continue to receive queued messages even when going offline.

Subscription A queue-based destination that receives messages from one or more topic destinations based on a topic name pattern and content-based matching. Subscriptions are analogous with queues in most ways except that they are never published to or selected on directly. Subscriptions may be private or shared, persistent or temporary. Subscriptions may be defined explicitly using the Subscription class, or defined automatically, when starting a consumer on a topic destination.

Consumer A client that requests messages from a destination.

Producer A client that publishes messages to a destination.

Virtual_host A collection of destinations and associated subscriptions, consumers, etc. Virtual hosts are independent server domains that share a common authentication and encryption environment.

Realm A group of destinations covered by a single security policy and access control. Applications ask for access to realms, rather than to individual destinations.

Streaming The process by which the server will send messages to the client at a pre-arranged rate.

Staging The process by which a peer will transfer a large message to a temporary holding area before formally handing it over to the recipient. This is how AMQP/Fast implements restartable file transfers.

Synchpoint A synchronous confirmed step in what is otherwise an asynchronous exchange of methods.

Out-of-band_transport The technique by which data is exchanged outside the network connection. For example, two peers can exchange frames across a TCP/IP connection and then switch to using shared memory if they discover they are on the same system.

Zero_copy The technique of transferring data without copying it to or from intermediate buffers. Zero copy requires that the protocol allow the transfer of data as opaque blocks, which AMQP/Fast does.

Assertion A condition that must be true for processing to continue.

Exception A failed assertion, handled by closing either the channel or the connection.

3.1.1.8 Limitations

These limitations are formally part of the AMQP/Fast specifications:

- Number of channels per connection, normal frames: 255.
- Number of channels per connection, extended frames: 64K-1.
- Size of a short string: 0 to 255 octets.
- Size of a long string: 0 to 64K-1 octets.
- Size of a normal frame: 0 to 64K-1 octets.
- Size of an extended frame: 0 to 2e64-1 octets.
- Number of possible content domains: 255.
- Number of protocol classes: 255 per peer.
- Number of methods: 255 per protocol class.
- Number of methods sent between synchpoints: 4G.

3.1.2 Data Type Representation

3.1.2.1 Goals and Principles

We use a small set of basic data types that are guaranteed to work on all platforms and be easily implemented in all languages. More sophisticated data types can be packaged using the basic AMQP/Fast data types.

3.1.2.2 Formal Grammar for AMQP/Fast Fields

This formal grammar defines the AMQP/Fast data types:

```
amqp-field          = BIT
                    / OCTET
                    / short-integer
                    / long-integer
                    / long-long-integer
                    / short-string
                    / long-string
                    / field-table
short-integer       = 2*OCTET
long-integer        = 4*OCTET
long-long-integer   = 8*OCTET
short-string        = OCTET *string-char
string-char         = %d1 .. %d255
long-string         = short-integer *OCTET
field-table         = long-integer *field-value-pair
field-value-pair    = field-name field-value
field-name          = short-string
field-value         = 'S' short-string
                    / 'L' long-string
                    / 'I' signed-integer
                    / 'D' decimal-value
                    / 'T' time-stamp
                    / 'F' field-table
signed-integer      = 4*OCTET
decimal-value       = decimals long-integer
decimals            = OCTET
time-stamp          = long-long-integer
```

3.1.2.3 Integers

AMQP/Fast defines these integer types:

- Bit values, which are accumulated into octets.

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

3.1.2.4 Strings

All strings are variable-length and represented by an integer length followed by zero or more octets of data. AMQP/Fast defines these string types:

- Short strings, stored as a 1-octet length followed by zero or more octets of data. Short strings are capable of carrying UTF-8 data, and may not contain binary zero octets. In the current version of the protocol short strings may only contain US-ASCII (ISO-8859-1) characters.
- Long strings, stored as a short integer length followed by zero or more octets of data. Long strings can contain any data.

3.1.2.5 Field Tables

Field tables are long strings that contain 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 short binary string, a long signed integer, a decimal, a date and/or time, or another field table.

3.1.3 Negotiating a Connection

3.1.3.1 Goals and Principles

Negotiation means that one party in a discussion declares an intention or capability and the other party either acknowledges it, modifies it, or rejects it. In AMQP/Fast, we negotiate a number of specific aspects of the protocol:

1. The actual protocol and version.
2. Encryption arguments and the authentication of both parties.
3. Operational constraints.

3.1.3.2 Protocol and Version

An AMQP client and server agree on a protocol and version using this negotiation model:

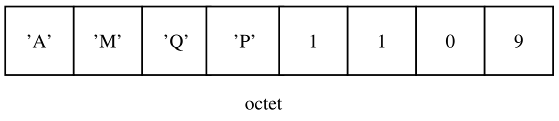
1. The client opens a new socket connection on a well-known or configured port and and sends an initiation sequence consisting of the text "AMQP" followed by a protocol ID (2 decimal digits) and a protocol version (2 decimal digits holding the high and low version numbers).
2. The server either accepts or rejects the initiation sequence. If it rejects the initiation sequence it closes the socket. Otherwise it leaves the socket open and implements the protocol accordingly.

Protocol grammar:

```
protocol-header     = 'AMQP' protocol-id protocol-version
protocol-id         = protocol-class protocol-instance
protocol-class      = OCTET
protocol-instance   = OCTET
protocol-version    = protocol-major protocol-minor
protocol-major      = OCTET
protocol-minor      = OCTET
```

For AMQP/Fast these are the correct values:

- protocol-class = 1 (this is the class of all AMQP/Fast protocols)
- protocol-instance = 1 (this is the instance of AMQP/Fast over TCP/IP)
- protocol-major = 0
- protocol-minor = 9



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.

3.1.3.3 Encryption and Authentication

AMQP/Fast uses the SASL architecture for security. SASL encapsulates TLS, GSSAPI, Kerberos, and other encryption and authentication technologies.

Security is negotiated between server and client as follows:

1. The server sends a challenge to the client. The challenge lists the security mechanisms that the server supports.
2. The client selects a suitable security mechanism and responds to the server with relevant information for that security mechanism.
3. The server can send another challenge, and the client another response, until the SASL layer at each end has received enough information.
4. The server can now use the selected security mechanism and authenticated client identity to perform access controls on its data and services.

The "relevant information" is an opaque binary blob that is passed between the SASL layers embedded in the client and in the server.

SASL gives us the freedom to replace the security libraries with better (more secure or faster) technologies at a later date without modifying the protocol, the client, or the server implementations.

3.1.3.4 Client and Server Limits

The protocol allows the client and server to agree on limits to ensure operational stability. 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.

Rather than fix the limits in the protocol, we negotiate them. This lets us tune the protocol dynamically for different types of efficiency, e.g. minimal memory consumption vs. maximum performance.

Limits are negotiated 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.

3.1.3.5 Identification and Capabilities

During the negotiation of limits the peers MUST exchange this mandatory information about themselves:

- The specific content domains that they support (see later).

A peer MAY use this information to adapt its behaviour appropriately.

And the peers MAY exchange this optional information about themselves:

- Their public product name, for logging and tracking.
- Their version number.
- The platform they are running on.

For security reasons, all these pieces of information are optional and a highly-security conscious peer MAY choose to provide some or none of them. A peer MUST NOT use this information to adapt its behaviour.

3.1.4 Multiplexing and Pipelining

3.1.4.1 Goals and Principles

Connection multiplexing allows multiple threads of communication to share a single socket connection. This is valuable when threads are short-lived since the cost of opening and closing TCP/IP sockets can be relatively high.

Pipelining means that each peer can send frames asynchronously without waiting for the recipient to acknowledge each one. This is an important facility because it greatly improves performance.

3.1.4.2 Multiplexing Design

AMQP/Fast uses the concept of "channel" to carry bi-directional streams of communication between peers. AMQP/Fast provides methods to open, use, and close channels. Large messages are broken into smaller frames so that channels get fair (round-robin) use of the socket connection.

3.1.4.3 Pipelining Design

Our design is based on these principles:

1. Most operations succeed. We can thus optimise traffic in many cases by stating that "no response" means "successful".
2. Channels represent serial streams, where the order of methods and their acknowledgements is stable.
3. In general we want to send data as fast as possible in an asynchronous fashion. Where necessary, we can implement windowing and throttling at a higher level.

These principles provide the basis for a simple and efficient pipelining design based on "selective synchronisation". Some methods are explicitly synchronous - a specific request is always paired with a specific reply unless there is an error - and other methods are asynchronous unless the sender asks for a synchronisation response.

The recipient indicates an error by closing the channel. The close method includes the ID of the method that caused the error.

Methods on a specific channel are processed strictly in order. Thus the client can set "synchronisation points" by asking for a confirmation, and handle errors unambiguously (all methods up to but not including the failed method will have succeeded).

3.1.5 The Framing Design

3.1.5.1 Goals and Principles

Framing is the part of the protocol where we define how data is sent "to the wire". The framing design used in AMQP/Fast is designed to be compact, easy and very fast to parse, extensible, and robust.

The key parts of this design are:

- How the transport layer carries data.
- How we delimit frames on the connection.
- How we multiplex frames on the connection.
- How different types of frame carry data.
- How we handle out-of-band transport.
- How we support future protocols (e.g. IPv6).

3.1.5.2 The Transport Layer

We assume a reliable stream-oriented network transport layer (TCP/IP or equivalent). If an unreliable transport layer is used, we assume that the AMQP/Fast frames would be wrapped with additional traffic-control information such as windowing.

AMQP/Fast explicitly excludes any support for traffic control but does not disallow this to be implemented in an additional layer. We may at a future point provide protocol wrappers that wrap AMQP/Fast frames with traffic management to allow reliable transfer over multicast and point-to-point UDP.

3.1.5.3 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 adds considerable overhead to the protocol due to the opening and closing of connections.
- Adding frame delimiters to the stream. This is used in protocols such as SMTP but has the disadvantage that the stream must be parsed to find the delimiters. This makes implementations of the protocol slow.
- Counting the size of frames and sending the size in front of each frame. This is the fastest approach, and our choice.

Each frame is thus sent as a "frame header" which contains the frame size, followed by a "frame body" of the specified size (the header is not counted). Frames can carry methods and other data. To read a frame we use this logic:

1. Read and decode the frame header to get the size of the frame body.
2. If the frame body is larger than the agreed limit, close the connection with a suitable error reply code.

3. Otherwise, read the specified number of octets into a frame buffer.
4. Decode the frame buffer as needed.

To write a frame we use this logic:

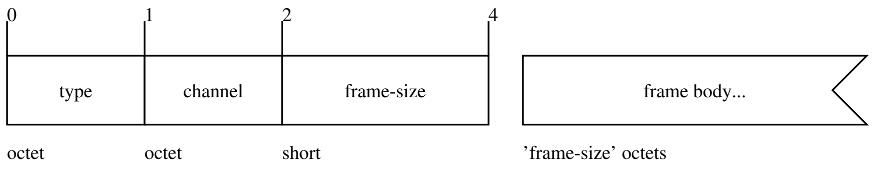
1. Marshal the frame buffer as needed.
2. Encode the length of this buffer into a frame header and write it.
3. Write the frame buffer contents, being the frame body.

Guidelines for implementors:

- A peer MUST write the frame body immediately after the frame header unless the body size is zero, or out-of-band transport is being used.

3.1.5.4 The Frame Header Wire Format

All frames start with a 4-octet header composed of a one-octet type indicator and a 3-octet size indicator:



AMQP/Fast defines several types of frame, with different type indicators:

- 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.
- Type = 9, "META": meta frame.

The channel number is 0 for all frames which are global to the connection, and 1 .. 255 for frames that refer to specific channels (1 .. 64K-1 in extended frames).

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.

3.1.5.5 Method Frames

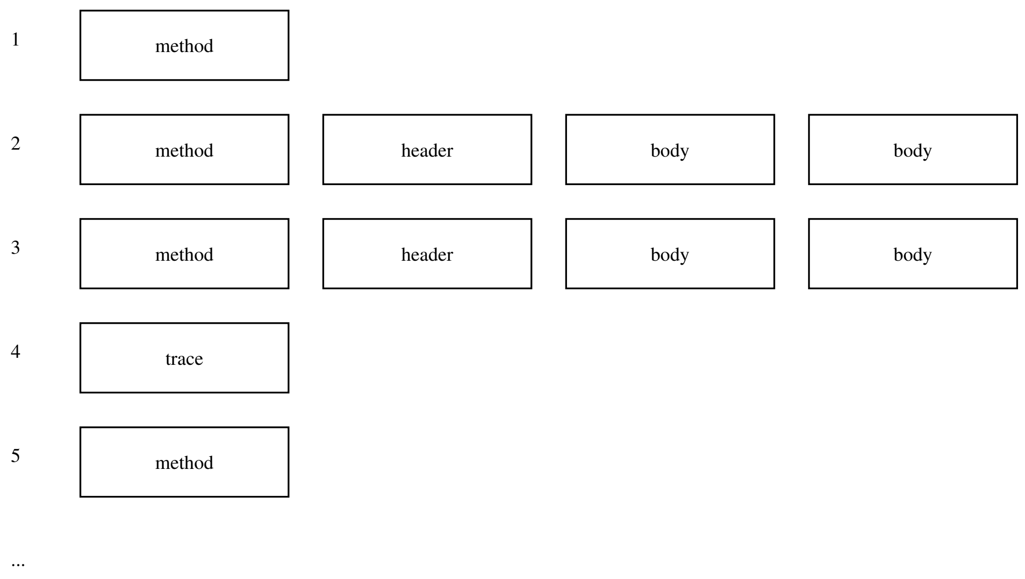
The method frame design is intended to be fast to read and write, and compact when only partly filled.

Method frames (also called "methods" in the following discussion, for brevity) are invariant. They contain no conditional or repeated fields. Methods are constructed out of a constant series of AMQP/Fast data fields (bits, integers, strings and string tables). Thus the marshalling code can be easily generated, and can be very rapid. In the OpenAMQ server and clients this code is generated directly from the protocol specifications.

3.1.5.6 Content Frames

Certain methods signal to the recipient that there is content arriving on the connection. Simple content consists of a header frame followed by zero or more body frames. Structured content may consist of a series of header frames followed by the appropriate body frames.

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



There is a specific reason for placing content body in distinct frames as compared to including the header and body in the method. We want to 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.

Content frames on a specific channel form an strict list. That is, they may be mixed with frames of different types or on different channels, but two contents may not be mixed or overlapped on a single channel.

3.1.5.7 Out-of-Band Transport Frames

Method, content header, and content body frames can be sent using out-of-band transport. The frame header is sent on the normal connection but the frame body is sent via another mechanism. The specific out-of-band transport used, and its configuration, is defined when a channel is opened.

3.1.5.8 Trace Frames

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

Both server and client MAY send trace frames at any point in the connection after protocol negotiation and before a Connection.Close method.

The semantics and structure and of trace frames including the channel number are not formally defined by AMQP/Fast and implementations MUST NOT assume any interoperability with respect to trace frames unless and until formal standards are defined for these.

3.1.5.9 Heartbeat Frames

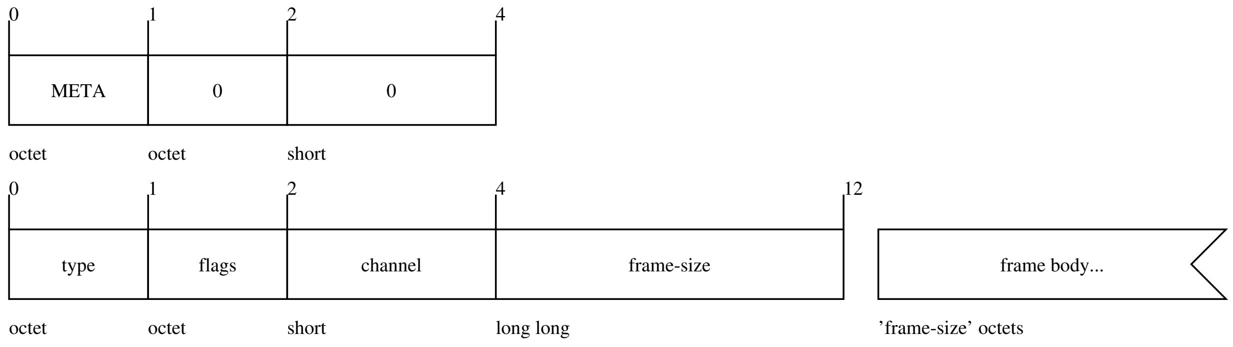
Although TCP/IP guarantees that data is not dropped or corrupted it can be slow to detect that a peer process has gone "offline". The heartbeat frame allows peers to detect network failure rapidly. This frame has no body.The peers negotiate heartbeat parameters at the start of a connection.

3.1.5.10 Meta Frames

Meta frames are used to send frames of 64KB or larger (the size of a normal frame body is limited to 64KB-1 octets), and to use channel numbers above 255. Meta frames are designed to support high-capacity out-of-band transports and IPv6 (which has a 4GB frame limit, although ethernet still cannot handle frames larger than 12000 bytes).

3.1.5.11 Extended Frame Header Wire Format

A meta frame is followed by an "extended frame header" with a short integer channel number and a long long size:



The frame body is not affected in any way by the presence or absence of a meta frame, except that the frame body MAY be larger than 64KB after a meta frame.

3.1.6 The Abstract Content Model

AMQP/Fast uses an abstract content model that has these goals and features:

- It supports tree-structured content in which each content can contain further content ("child-content"), to any level.
- It provides multiple content types to allow optimal encoding for different applications.
- It supports content bodies of any size from zero octets up to infinity.
- Content body can be read and written directly from application memory with no formatting or copying ("zero copy").
- Content headers are sent before content body so that the recipient can selectively discard content that it does not want to process.
- Content body is sent in separate frames to support the AMQP/Fast channel multiplexing model.

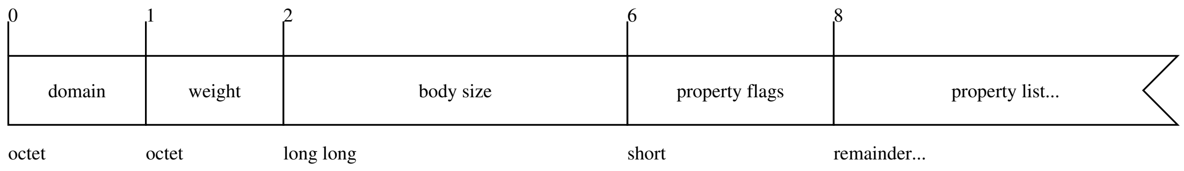
3.1.6.1 Formal Grammar for Content Frames

This is the formal grammar for the AMQP/Fast content model:

```
content         = header-frame child-content *body-frame
header-frame    = HEADER channel frame-size header-payload
channel         = octet
frame-size      = short-integer
header-payload  = class weight body-size
                  property-flags property-list
content-class   = OCTET
content-weight  = OCTET
body-size       = long-long-integer
property-flags  = 15*BIT %b0 / 15*BIT %b1 property-flags
property-list   = amqp-field
child-content   = weight*content
body-frame      = BODY channel frame-size body-payload
body-payload    = *OCTET
```

3.1.6.2 Content Header Frame Wire Format

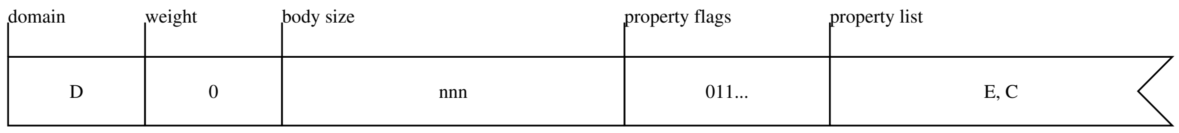
A content header frame has this format:



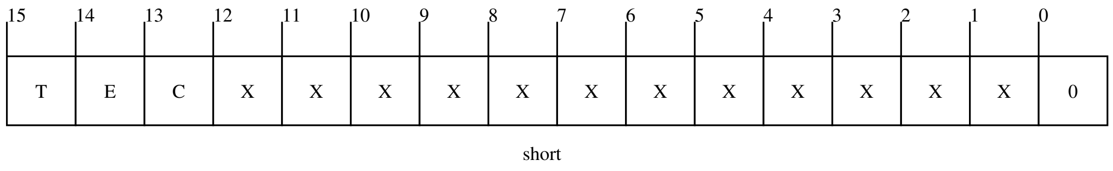
- The content class specifies the set of properties (with a predefined syntax and semantics) that the header frame may hold. The standard AMQP/Fast content classes are defined later.
- The weight field specifies whether the content is structured or not. Unstructured content has a weight of zero. Structured content has a weight of 1 to 255. This is the number of child-contents that the content contains.
- The body size is a 64-bit value that defines the total size of the body content. It may be zero, indicating that there will be no body frames.
- The property flags are a bit array that indicates the presence or absence of each property value.
- The property values are class-specific data fields. The properties are formatted as AMQP/Fast data types (integers, strings, field tables).

The set of properties for a content class can be any size. The first two properties for all content classes are the content MIME type and the content encoding. Following these, each content class has a specific set of properties defined in a strict order from most to least significant.

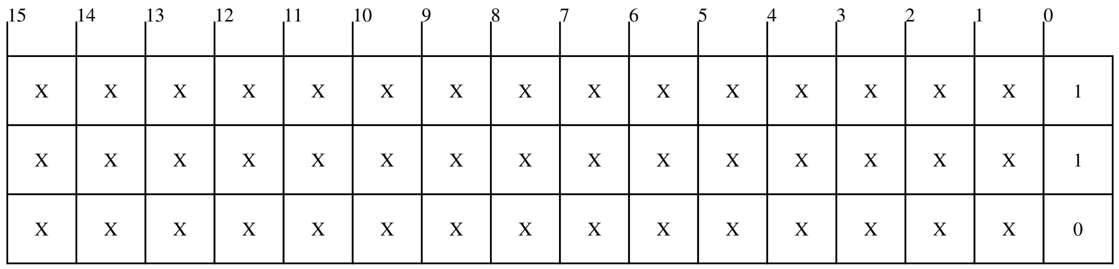
As an example, we take an imaginary content class "D" which has three properties, T, E and C. Imagine a simple content that has properties E and C but not T. The header frame will be formatted thus:



The property flags are ordered from high to low, the first property being indicated by bit 15, and so on:

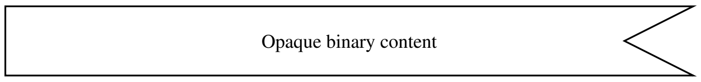


The property flags can specify more than 16 properties. If bit 0 is set, it indicates that a further property flags field follows. There can be several property flag fields in succession:



3.1.6.3 Content Body Frame Wire Format

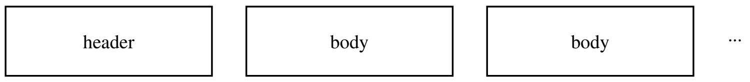
A content body frame has this format:



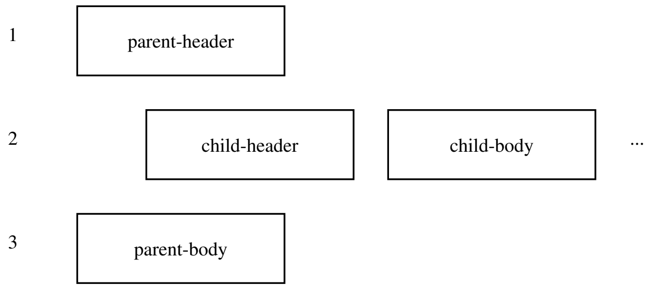
Where the size of the frame is defined in the frame header.

3.1.6.4 Structured Content

The "weight" field in the content header frame indicates whether the content has child-contents or not. In the simplest case (weight = zero) content consists of a header frame plus zero or more body frames:



If the content has child-contents, these are inserted between the header and body of the parent content:



This organisation means that a principle content (e.g. a very large video file) can have child content (e.g. subtitles and menus) which will be sent before the main content. If a more specific order of content is needed, it can be done by defining a top-level content that has no body, only a header.

3.1.6.5 Multiplexed Channel Content

Content consists of one or more frames. The recipient of content that has been sent in multiple frames can unambiguously reconstruct the content by implementing the content grammar defined above, on a per-channel basis.

Guidelines for implementors:

- Content frames MUST be sent on a single channel, and MAY be intermixed with non-content frames (method, trace, and heartbeat frames).
- Content frames MUST NOT be intermixed with frames of a different content.
- A peer MUST NOT make any assumptions about the minimum or maximum size of a body frame except that the minimum size is zero bytes and the maximum size is the smallest of either the negotiated maximum frame size or the remaining expected of content body (body-size minus amount of body data received so far).
- Empty body frames are permitted but a peer may not assign any special meaning to these: it MUST discard them with no side-effects of any kind.

3.1.6.6 Content Classes

The content class is not intended to provide data typing or encoding information. Its purpose is to allow functional clarity in the property sets and methods that we define per class of message.

Content classes turn AMQP/Fast from an abstract content-carrying protocol into a functional tool by providing data and operations that are directly and clearly mapped to the kinds of API that an application programmer needs to see.

AMQP/Fast defines these standard content classes:

- JMS: the content is a JMS-compliant message.
- File: the content is a file.
- Stream: the content is a streamed message.

Note that the ID of each content class is the same as the ID of the protocol class (JMS, File, Stream) that supports it.

Each class has a specific access model, though they all share the basic AMQP queueing and routing mechanisms including topic multipliers. These classes are defined in the next sections. This model lets us extend AMQP/Fast into new functional areas by defining new content classes and appropriate methods, without breaking interoperability.

Guidelines for implementors:

- The client MAY support any or all of the defined content classes.
- The server MUST support at least one of the defined content classes.

3.1.6.7 Content Type and Encoding

AMQP/Fast enforces MIME-compliance on all content classes using two standard properties that are the first defined in all content classes:

- ContentType (tiny string) - the MIME content type of the message body. The default value is "application/octet-stream".

- ContentEncoding (tiny string) - The content encoding of the message body. The default value is "binary".

Guidelines for implementors:

- The server SHOULD not modify the ContentType and ContentEncoding for content moved between producing and consuming clients.
- The server MAY set the ContentType and ContentEncoding properties for content produced by the server or by applications embedded in the server.

3.1.7 The Class/Method Model

3.1.7.1 Goals and Principles

AMQP/Fast allows two peers to connect and exchange data. There are several types of data and several ways of exchanging this data. The combinations rapidly get complex. Worse, it is not always clear which replies depend on which requests, since the exchange of data is often asynchronous, pipelined, and generally hard to follow.

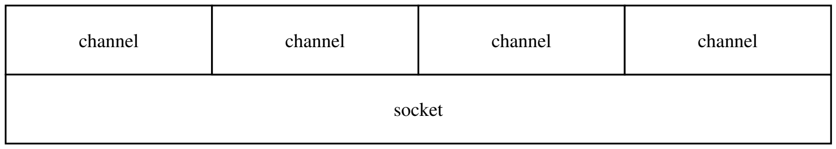
Rather than attempt to formalise the exact flow of data from client to server and back, we use an metaphor taken from the software world, namely the object-oriented metaphor of "classes" and "methods".

We build this as follows:

1. Each peer supports a set of classes. These classes have names and cover specific areas of functionality. A peer can support all the AMQP/Fast classes, or a subset of them. Since the operational functionality of the protocol depends on the classes, the operational functionality of a particular implementation is clear to see.
2. Each class supports a set of methods. These methods have names and provide specific functionalities. A peer can support all the AMQP/Fast methods in a particular class, or a subset of them. As with classes, this unambiguously defines the operational functionality of a peer.
3. The methods are either synchronous or asynchronous. A synchronous method replies immediately. An asynchronous method replies at some later time and may send multiple replies. A method "replies" by invoking the appropriate method in the requesting peer. A request and reply always involves two methods: one to implement the request and one to implement the reply.
4. Each method is either a client method, server method, or provided by both client and server). A particular method in a particular class always has an identical behaviour whether it is in a "client" or "server".
5. A peer can thus combine the roles of client and server in various degrees by implementing the necessary classes and methods. At any point the operational functionality of the peer can be unambiguously defined by the methods that it supports.

3.1.7.2 Client and Server Roles

The AMQP/Fast protocol is pseudo-symmetrical with client-initiated connections. We can visualise the protocol as governing two levels of connection - network connections (sockets) and virtual connections (channels):



Each connection (socket and channel) has a clear client role - the peer that initiated the connection, and a clear server role - the peer that accepted the connection. For both network and virtual connections the general protocol model is the same:

1. The client requests the server to open connection.
2. The server responds and the peers negotiate the connection.
3. The client and the server exchange information.
4. Either peer requests to close the connection.
5. The other peer responds and they negotiate the shut-down.

AMQP/Fast forsees two main architectures that use this model:

1. Client-server, in which the same peer acts as client in all connections. This is the "usual" architecture.
2. Peer-to-peer, in which one peer acts as client in the network connection but the peers can take either role in the virtual connection.

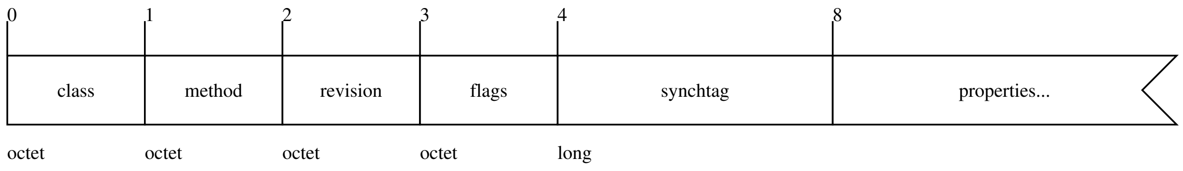
The peer-to-peer architecture requires that both peers can act as both client and server.

Guidelines for implementors:

- A peer MUST support at least either the client or server role.
- A peer MAY support both roles.

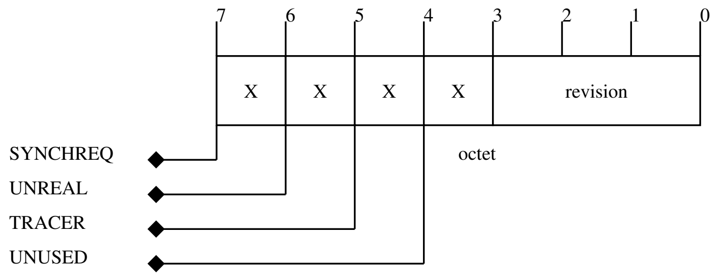
3.1.7.3 Method Frame Body Wire Format

A method frame body has this format:



- The class and method are protocol-constant values.
- The revision number indicates a revision of the method with new or modified properties.
- The flags provide a set of operational indicators.
- The synchtag is an arbitrary value that the sender chooses to identify the method instance.
- The properties are a set of AMQP/Fast fields that specific to each method.

The flags are defined as follows:



- 7 = SYNCHREQ - if 1, requests a ”synchpoint”, turning an asynchronous method into a synchronous one. Has no effect on a synchronous method.
- 6 = UNREAL - if 1, requests an ”interface test”, in which the recipient responds with an echo of the method but does not otherwise execute the method.
- 5 = TRACER - if 1, requests a ”distributed trace”, in which the recipient will trace all responses to this and other clients which the method provokes.
- 4 = UNUSED - this bit is not used.
- 3-0 = revision - must be zero in this protocol.

Guidelines for implementors:

- The class MUST be one of the standard class numbers defined below.
- The method MUST be one of the methods defined for the class.
- The revision MUST be zero in major releases of the protocol. It is used to indicate revised methods in minor releases of the protocol. A peer MAY support multiple revisions of a protocol. A peer MUST support the specific method revisions defined in the protocol minor version number agreed during connection negotiation.
- Synchtags SHOULD BE unique since the last synchpoint (we explain synchpoints later).
- Bits 2 to 0 of the flags octet are reserved for future use and MUST be zero.

3.1.7.4 The Synchpoint Function

A client can request a synchpoint when sending an asynchronous request method. The server responds with a Channel SYNCH method when it finishes processing the request method. Synchpoints are only sent if the method completes successfully. If the method fails, the server replies with a Channel.Close or Connection.Close depending on the severity of the error.

Clients can use synchpoints to force occasional synchronisation or to force a fully-synchronous dialogue. E.g. A client can be designed to use synchpoints at every asynchronous method, which turns the entire protocol into a synchronous protocol. This may be simpler for some scenarios than a normal asynchronous exchange of methods.

3.1.7.5 The Interface Test Function

A peer can test the capability of the other peer by sending methods with the UNREAL flag set. When a peer receives a message with this flag set it de-marshals the frame, reconstructs it with the same fields, and sends it back to the requesting peer. We assume that a specially designed peer, a ”monitor client” will perform the tests.

The goal of the UNREAL flag is to allow a monitor client to determing what methods a peer supports.

3.1.7.6 The Distributed Trace Function

A distributed trace is a temporary and single-threaded trace of all messages produced by a specific server method. A distributed trace will last as long as the effects of the original method. There is no functionality to end a distributed trace, nor to distinguish more than one trace at once.

3.1.8 Summary of Classes and Methods

3.1.8.1 Goals and Principles

We aim to provide a clear and extensible organisation of the classes and methods that a peer provides. Classes and methods are numbered but also have constant names (shown in uppercase). The names are chosen to be consistent across classes, so a similar method in different classes has the same name and may have similar arguments where appropriate. The full definition of each class and method is provided later.

We define methods as being either:

- a synchronous request ("syn request"). The sending peer SHOULD wait for the specific reply method, but MAY implement this asynchronously.
- a synchronous reply ("syn reply for XYZ").
- an asynchronous request or reply ("async"). The sending peer will usually not expect a synchronous reply but MAY set the SYNCHREQ flag to request a synchpoint.

3.1.8.2 Protocol Classes

AMQP/Fast defines these protocol classes:

- Connection: work with socket connections (ID = 1).
- Channel: work with channels (ID = 2).
- Access: work with access tickets (ID = 3).
- Destination: work with dynamic destinations (ID = 4).
- Subscription: work with subscriptions (ID = 5).
- Jms: work with JMS content (ID = 6).
- File: work with file content (ID = 7).
- Stream: work with streaming content (ID = 8).
- Tx: work with standard transactions (ID = 9).
- Dtx: work with distributed transactions (ID = 10).
- Test: test functional primitives of the implementation (ID = 11).

3.1.8.3 The Connection Class

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.

This is the formal grammar for the class:

```
connection          = open-connection *use-connection close-connection
open-connection     = C:protocol-header
                      S:START C:START_OK
                      *challenge
                      S:TUNE C:TUNE_OK
                      C:OPEN S:OPEN_OK
challenge           = S:SECURE C:SECURE_OK
use-connection      = S:UNKNOWN
                    / C:UNKNOWN
                    / channel
close-connection    = C:CLOSE S:CLOSE_OK
                    / S:CLOSE C:CLOSE_OK
```

This class contains the following server methods:

- Connection.Start Ok - select security mechanism (ID = 2) (sync reply for Start)
- Connection.Secure Ok - seurity mechanism response (ID = 4) (sync reply for Secure)
- Connection.Tune Ok - negotiate connection tuning parameters (ID = 6) (sync reply for Tune)
- Connection.Open - open a path to a virtual host (ID = 7) (sync request)
- Connection.Unknown - signal that an interface test method has failed (ID = 9) (async)
- Connection.Close - request a connection close (ID = 10) (sync request)
- Connection.Close Ok - confirm a connection close (ID = 11) (sync reply for Close)

This class contains the following client methods:

- Connection.Start - start connection negotiation (ID = 1) (sync request)
- Connection.Secure - seurity mechanism challenge (ID = 3) (sync request)
- Connection.Tune - propose connection tuning parameters (ID = 5) (sync request)
- Connection.Open Ok - signal that the connection is ready (ID = 8) (sync reply for Open)
- Connection.Unknown - signal that an interface test method has failed (ID = 9) (async)
- Connection.Close - request a connection close (ID = 10) (sync request)
- Connection.Close Ok - confirm a connection close (ID = 11) (sync reply for Close)

Connection.Start This method starts the connection negotiation process by telling the client the protocol version that the server proposes, along with a list of security mechanisms which the client can use for authentication.

```
version_major      octet       #  negotiated protocol major version
version_minor      octet       #  negotiated protocol major version
mechanisms         shortstr    #  available security mechanisms
```

Connection.Start Ok This method selects a SASL security mechanism. AMQP/Fast uses SASL (RFC2222) to negotiate authentication and encryption.

```
mechanism          shortstr    #  selected security mechanism
```

Connection.Secure The SASL protocol works by exchanging challenges and responses until both peers have received sufficient information to authenticate each other. This method challenges the client to provide more information.

```
challenge          longstr     #  security challenge data
```

Connection.Secure Ok This method attempts to authenticate, passing a block of SASL data for the security mechanism at the server side.

```
response          longstr     #  security response data
```

Connection.Tune This method proposes a set of connection configuration values to the client. The client can accept and/or adjust these.

```
frame_max         long        #  maximum frame size
channel_max       short       #  maximum number of channels
access_max        short       #  maximum number of access tickets
consumer_max      short       #  maximum consumers per channel
heartbeat         short       #  desired heartbeat delay
txn_limit         short       #  maximum transaction size
jms_support       bit         #  JMS content is supported?
file_support      bit         #  file content is supported?
stream_support    bit         #  stream content is supported?
```

Connection.Tune Ok This method sends the client's connection tuning parameters to the server. Certain fields are negotiated, others provide capability information.

```
frame_max         long        #  maximum frame size
channel_max       short       #  maximum number of channels
ticket_max        short       #  maximum number of access tickets
heartbeat         short       #  desired heartbeat delay
jms_support       bit         #  JMS content is supported?
file_support      bit         #  file content is supported?
stream_support    bit         #  stream content is supported?
prefetch_max      long        #  maximum prefetch size for connection
```

Connection.Open This method opens a path to a virtual host on the server. The virtual host is a collection of destinations, and acts to separate multiple application domains on the server.

```
virtual_path      shortstr    #  path value virtual server path
client_id         shortstr    #  client identifier
```

Connection.Open Ok This method signals to the client that the connection is ready for use.

```
client_id         shortstr    #  assigned client identifier
```

Connection.Unknown This method signals that an interface test method has failed. This may happen after a method is sent with the UNREAL flag set.

```
class             octet       #  failing method class
method            octet       #  failing method ID
synchtag          short       #  failing method synchtag
reply_code        short       #  reply code from server reply code
reply_text        shortstr    #  localised reply text localised reply text
```

Connection.Close This method indicates that the sender wants to close the connection. This may be due to internal conditions (e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class, method id, and synchtag of the method which caused the exception.

```
reply_code        short       #  reply code from server reply code
reply_text        shortstr    #  localised reply text localised reply text
class             octet       #  failing method class
method            octet       #  failing method ID
synchtag          short       #  failing method synchtag
```

Connection.Close Ok This method confirms a Connection.Close method and tells the recipient that it is safe to release resources for the connection and close the socket. This method has no fields apart from the standard method header.

3.1.8.4 The Channel Class

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.

This is the formal grammar for the class:

```
channel             = open-channel *use-channel close-channel
open-channel        = C:OPEN S:OPEN_OK
use-channel         = C:FLOW
                    / S:FLOW
                    / S:SYNCH
                    / access
                    / destination
                    / subscription
                    / jms
                    / file
                    / stream
                    / tx
                    / dtx
                    / test
close-channel       = C:CLOSE S:CLOSE_OK
                    / S:CLOSE C:CLOSE_OK
```

This class contains the following server methods:

- Channel.Open - open a channel for use (ID = 1) (sync request)
- Channel.Flow - enable/disable flow from peer (ID = 3) (async)
- Channel.Close - request a channel close (ID = 4) (sync request)
- Channel.Close Ok - confirm a channel close (ID = 5) (sync reply for Close)

This class contains the following client methods:

- Channel.Open Ok - signal that the channel is ready (ID = 2) (sync reply for Open)
- Channel.Flow - enable/disable flow from peer (ID = 3) (async)
- Channel.Close - request a channel close (ID = 4) (sync request)
- Channel.Close Ok - confirm a channel close (ID = 5) (sync reply for Close)

Channel.Open Client asks server to open a new channel.

```
prefetch_max       long        #  maximum prefetch size for channel
out_of_band        shortstr    #  out-of-band settings for channel
tx_mode            octet       #  transaction mode for channel
```

Channel.Open Ok This method signals to the client that the channel is ready for use. This method has no fields apart from the standard method header.

Channel.Flow This method asks the peer to pause or restart the flow of content data. This is a simple flow-control mechanism that a peer can use to avoid oveflowing its queues or otherwise finding itself receiving more messages than it can process. Note that this method is not intended for window control.

```
flow_pause         bit         #  start/stop content frames
```

Channel.Close This method indicates that the sender wants to close the channel. This may be due to internal conditions (e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class, method id, and synchtag of the method which caused the exception.

```
reply_code         short       #  reply code from server reply code
reply_text         shortstr    #  localised reply text localised reply text
class              octet       #  failing method class
method             octet       #  failing method ID
synchtag           short       #  failing method synchtag
```

Channel.Close Ok This method confirms a Channel.Close method and tells the recipient that it is safe to release resources for the channel and close the socket. This method has no fields apart from the standard method header.

3.1.8.5 The Access Class

AMQP/Fast controls 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.

This is the formal grammar for the class:

```
access              = C:GRANT S:GRANT_OK
```

This class contains the following server methods:

- Access.Grant - request an access ticket (ID = 1) (sync request)

This class contains the following client methods:

- Access.Grant Ok - grant access to server resources (ID = 2) (sync reply for Grant)

Access.Grant This method requests an access ticket for an access realm. The server responds by granting the access ticket. If the client does not have access rights to the requested realm this causes a connection exception. Access tickets may be shared across channels within a connection and expire with the connection.

```
realm              shortstr    #  path value realm to work with
exclusive          bit         #  request exclusive access
query              bit         #  request query rights
publish            bit         #  request publish rights
consume            bit         #  request consume rights
subscribe          bit         #  request subscriber rights
dynamic            bit         #  request dynamic destination rights
purge              bit         #  request purge rights
```

Access.Grant Ok This method provides the client with an access ticket. The access ticket is valid within the current channel and for the lifespan of the channel.

```
ticket             short       #  access ticket granted by server
```

3.1.8.6 The Destination Class

Destinations are queues or topics, capable of storing and routing all content classes, that are created at runtime rather than as server configured objects. A dynamic destination is normally peristent - i.e. it continues to exist and accept messages even when the creating client has disconnected, or the server has restarted. This class lets clients create and manage dynamic destinations.

This is the formal grammar for the class:

```
destination         = C:DEFINE S:DEFINE_OK
                    / C:QUERY  S:QUERY_OK
                    / C:PURGE  S:PURGE_OK
                    / C:CANCEL S:CANCEL_OK
```

This class contains the following server methods:

- Destination.Define - create or verify a dynamic destination (ID = 1) (sync request)
- Destination.Query - query status of destination (ID = 3) (sync request)
- Destination.Purge - purge a destination (ID = 5) (sync request)
- Destination.Cancel - cancel a dynamic destination (ID = 7) (sync request)

This class contains the following client methods:

- Destination.Define Ok - confirms a destination definition (ID = 2) (sync reply for Define)
- Destination.Query Ok - provide status of destination (ID = 4) (sync reply for Query)
- Destination.Purge Ok - confirms a destination purge (ID = 6) (sync reply for Purge)
- Destination.Cancel Ok - confirm cancellation of a dynamic destination (ID = 8) (sync reply for Cancel)

Destination.Define This method creates or reconfigures a dynamic destination. Various fields in this method let the client control the persistence of the destination. For example, one can implement a JMS-style temporary destination by setting both the private and auto-cancel fields. A client can work with an existing dynamic destination without using this method.

```
ticket             short       #  access ticket granted by server
service_type       octet       #  destination service type
destination        shortstr    #  destination name
template           shortstr    #  destination template
private            bit         #  request private destination
auto_cancel        bit         #  auto-cancel when finished
```

Destination.Define Ok This method confirms a Define method and confirms the name of the destination, essential for automatically-named destinations.

```
destination        shortstr    #  destination name name of destination
message_count      long        #  number of messages in queue destination
```

Destination.Query This method queries the status of a destination. This can be a dynamic destination or a configured destination, and it can be a queue or a topic as defined by the service-type field.

```
ticket             short       #  access ticket granted by server
service_type       octet       #  destination service type
destination        shortstr    #  destination name
```

Destination.Query Ok This method returns the status of the destination.

```
service_type       octet       #  destination service type
destination        shortstr    #  destination name name of destination
message_count      long        #  number of messages in queue destination
consumer_count     long        #  number of consumers
subscriber_count   long        #  number of subscribers
```

Destination.Purge This method removes all messages from a queue destination or all messages from the subscriptions for a topic. It does not cancel consumers or subscriptions. Purged messages are deleted without any formal ”undo” mechanism.

```
ticket             short       #  access ticket granted by server
service_type       octet       #  destination service type
destination        shortstr    #  destination name name of destination
```

Destination.Purge Ok This method confirms the purge of a destination.

```
message_count      long        #  number of messages purged
```

Destination.Cancel This method cancels (deletes) a dynamic destination. When a queue is cancelled, any pending messages are sent to a dead-letter queue if this is defined in the server configuration, and all consumers on the queue are cancelled. When a topic is cancelled, all subscriptions on that topic are also cancelled.

```
ticket             short       #  access ticket granted by server
service_type       octet       #  destination service type
destination        shortstr    #  destination name
ifempty            bit         #  cancel only if empty
```

Destination.Cancel Ok This method confirms the cancellation of a destination.

```
message_count      long        #  number of messages purged
```

3.1.8.7 The Subscription Class

Subscriptions are requests to receive messages from one or more topics. Subscriptions can be temporary or persistent, and private or shared. In the AMQ queuing and routing model, subscriptions are orthogonal with queues and provide very similar functionality except that a client can not send messages directly to a subscription, only via a topic. Subscriptions are capable of storing all content domains. This class provides methods for clients to create, manage, and share subscriptions.

This is the formal grammar for the class:

```
subscription        = C:DEFINE S:DEFINE_OK
                    / C:QUERY  S:QUERY_OK
                    / C:PURGE  S:PURGE_OK
                    / C:CANCEL S:CANCEL_OK
```

This class contains the following server methods:

- Subscription.Define - create or configure a persistent subscription (ID = 1) (sync request)
- Subscription.Query - query status of subscription (ID = 3) (sync request)
- Subscription.Purge - purge a subscription (ID = 5) (sync request)
- Subscription.Cancel - cancel a subscription (ID = 7) (sync request)

This class contains the following client methods:

- Subscription.Define Ok - confirms a subscription definition (ID = 2) (sync reply for Define)
- Subscription.Query Ok - provide status of subscription (ID = 4) (sync reply for Query)
- Subscription.Purge Ok - confirms a subscription purge (ID = 6) (sync reply for Purge)
- Subscription.Cancel Ok - confirm cancellation of a subscription (ID = 8) (sync reply for Cancel)

Subscription.Define This method creates or reconfigures a subscription. A subscription is normally peristent - i.e. it continues to exist and accept messages even when the creating client has disconnected, or the server has restarted. Various fields in this method let the client control the persistence of the subscription. A client can work with an existing subscription without using this method.

```
ticket             short       #  access ticket granted by server
subscription       shortstr    #  subscription name
template           shortstr    #  subscription template
topic_selector     shortstr    #  topic name or pattern
topic_seltype      octet       #  type of topic selector
message_selector   longstr     #  message selector
message_seltype    octet       #  type of message selector
private            bit         #  request private subscription
auto_cancel        bit         #  auto-cancel when finished
```

Subscription.Define Ok This method confirms a Subscription.Define method and provides the client with the name of the subscription in case of automatically named subscriptions. The server also reports the topic selector and number of messages for existing subscriptions.

```
subscription       shortstr    #  subscription name
topic_selector     shortstr    #  topic name or pattern
topic_seltype      octet       #  type of topic selector
message_count      long        #  number of messages in subscription
```

Subscription.Query This method queries the status of a subscription.

```
ticket             short       #  access ticket granted by server
subscription       shortstr    #  subscription name subscription name
```

Subscription.Query Ok This method returns the status of the subscription.

```
subscription       shortstr    #  subscription name
message_count      long        #  number of messages in subscription
consumer_count     long        #  number of consumers
```

Subscription.Purge This method removes all messages from a subscription. It does not cancel any consumers on the subscription.

```
ticket             short       #  access ticket granted by server
subscription       shortstr    #  subscription name
```

Subscription.Purge Ok This method confirms the purge of a subscription.

```
message_count      long        #  number of messages purged
```

Subscription.Cancel This method cancels a subscription. Note that cancellation implies a purge: any pending messages in the subscription are discarded. All consumers for the subscription are also cancelled.

```
ticket             short       #  access ticket granted by server
subscription       shortstr    #  subscription name
ifempty            bit         #  cancel only if empty
```

Subscription.Cancel Ok This method confirms the cancellation of a subscription.

```
message_count      long        #  number of messages purged
```

3.1.8.8 The Jms Class

The JMS class provides methods that support the standard JMS API. JMS messages have a specific set of properties that are required for interoperability with JMS providers and consumers. JMS messages and acknowledgements are subject to channel transactions.

This is the formal grammar for the class:

```
jms                 = C:CONSUME S:CONSUME_OK
                    / C:CANCEL
                    / C:PUBLISH content
                    / S:DELIVER content
                    / C:BROWSE ( S:BROWSE_OK content / S:BROWSE_EMPTY )
                    / C:ACK
                    / C:REJECT
```

This class contains the following server methods:

- Jms.Consume - start a destination consumer (ID = 1) (sync request)
- Jms.Cancel - end a destination consumer (ID = 3) (async)
- Jms.Publish - publish a message to a destination (ID = 4) (async, carries content)
- Jms.Browse - direct access to a destination (ID = 6) (sync request)
- Jms.Ack - acknowledge one or more messages (ID = 9) (async)
- Jms.Reject - reject an incoming message (ID = 10) (async)

This class contains the following client methods:

- Jms.Consume Ok - confirm a new consumer (ID = 2) (sync reply for Consume)
- Jms.Deliver - notify the client of a consumer message (ID = 5) (async, carries content)
- Jms.Browse Ok - provide client with a browsed message (ID = 7) (sync reply for Browse, carries content)
- Jms.Browse Empty - indicate no messages available (ID = 8) (sync reply for Browse)

Jms.Consume This method asks the server to start a "consumer", which is a temporary request for messages from a specific queue or topic subscription. Consumers last as long as the channel they were created on, or until they are cancelled.

```
ticket            short      #  access ticket granted by server
service_type      octet      #  destination service type
destination       shortstr   #  destination name
subscription      shortstr   #  subscription name
prefetch_size     short      #  prefetch window in octets
prefetch_count    short      #  prefetch window in messages
no_local          bit        #  do not receive own messages
auto_ack          bit        #  no acknowledgement needed
exclusive         bit        #  request exclusive access
message_selector  longstr    #  message selector
message_seltype   octet      #  type of message selector
auto_cancel       long       #  cancel consumer after N messages
```

Jms.Consume Ok This method provides the client with a consumer tag which it may use in methods that work with the consumer.

```
consumer_tag      short      #  server-assigned consumer tag
```

Jms.Cancel This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer.

```
consumer_tag      short      #  server-assigned consumer tag
```

Jms.Publish This method publishes a message to a specific destination. The message will be saved to the destination and distributed to any consumers when the transaction is committed.

```
ticket            short      #  access ticket granted by server
service_type      octet      #  destination service type
destination       shortstr   #  destination name
immediate         bit        #  assert immediate delivery
```

Jms.Deliver This method delivers a message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the Consume method, then the server responds with Deliver methods as and when messages arrive for that consumer.

```
delivery_tag      longlong   #  server-assigned delivery tag
redelivered       bit        #  signal a redelivered message
destination       shortstr   #  destination name
```

Jms.Browse This method provides a direct access to the messages in a queue or subscription using a synchronous dialogue that is designed for specific types of application where functionality is more important than performance.

```
ticket            short      #  access ticket granted by server
service_type      octet      #  destination service type
destination       shortstr   #  destination name
subscription      shortstr   #  subscription name
no_local          bit        #  do not receive own messages
auto_ack          bit        #  no acknowledgement needed
```

Jms.Browse Ok This method delivers a message to the client following a browse method. A browsed message will need to be acknowkedged, unless the auto-ack option was set to one.

```
delivery_tag      longlong   #  server-assigned delivery tag
redelivered       bit        #  signal a redelivered message
destination       shortstr   #  destination name
message_count     long       #  number of messages pending
```

Jms.Browse Empty This method tells the client that the queue or subscription has no messages available for the client. This method has no fields apart from the standard method header.

Jms.Ack This method acknowledges one or more messages delivered via the Deliver or Browse-Ok methods. The client can ask to confirm a single message or a set of messages up to and including a specific message.

```
delivery_tag       longlong    #  server-assigned delivery tag
multiple           bit         #  acknowledge multiple messages
```

Jms.Reject This method allows a client to reject a message. It can be used to cancel large incoming messages, or return unprocessable messages to their original queue or subscription.

```
delivery_tag       longlong    #  server-assigned delivery tag
requeue            bit         #  requeue the message
```

3.1.8.9 The File Class

The file class provides methods that support reliable file transfer. File messages have a specific set of properties that are required for interoperability with file transfer applications. File messages and acknowledgements are subject to channel transactions.

This is the formal grammar for the class:

```
file                = C:OPEN S:OPEN_OK
                    / S:OPEN C:OPEN_OK
                    / C:STAGE
                    / S:STAGE
                    / C:CONSUME S:CONSUME_OK
                    / C:CANCEL
                    / C:PUBLISH content
                    / S:DELIVER content
                    / C:BROWSE ( S:BROWSE_OK content / S:BROWSE_EMPTY )
                    / C:ACK
                    / C:REJECT
```

This class contains the following server methods:

- File.Consume - start a destination consumer (ID = 1) (sync request)
- File.Cancel - end a destination consumer (ID = 3) (async)
- File.Publish - publish a message to a destination (ID = 4) (async, carries content)
- File.Browse - direct access to a destination (ID = 6) (sync request)
- File.Ack - acknowledge one or more messages (ID = 9) (async)
- File.Reject - reject an incoming message (ID = 10) (async)

This class contains the following client methods:

- File.Consume Ok - confirm a new consumer (ID = 2) (sync reply for Consume)
- File.Deliver - notify the client of a consumer message (ID = 5) (async, carries content)
- File.Browse Ok - provide client with a browsed message (ID = 7) (sync reply for Browse, carries content)
- File.Browse Empty - indicate no messages available (ID = 8) (sync reply for Browse)

File.Consume This method asks the server to start a "consumer", which is a temporary request for messages from a specific queue or topic subscription. Consumers last as long as the channel they were created on, or until they are cancelled.

```
ticket             short       #  access ticket granted by server
service_type       octet       #  destination service type
destination        shortstr    #  destination name
subscription       shortstr    #  subscription name
prefetch_size      short       #  prefetch window in octets
```

```
prefetch_count     short       #  prefetch window in messages
no_local           bit         #  do not receive own messages
auto_ack           bit         #  no acknowledgement needed
exclusive          bit         #  request exclusive access
message_selector   longstr     #  message selector
message_seltype    octet       #  type of message selector
auto_cancel        long        #  cancel consumer after N messages
```

File.Consume Ok This method provides the client with a consumer tag which it may use in methods that work with the consumer.

```
consumer_tag       short       #  server-assigned consumer tag
```

File.Cancel This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer.

```
consumer_tag       short       #  server-assigned consumer tag
```

File.Publish This method publishes a message to a specific destination. The message will be saved to the destination and distributed to any consumers when the transaction is committed.

```
ticket             short       #  access ticket granted by server
service_type       octet       #  destination service type
destination        shortstr    #  destination name
immediate          bit         #  assert immediate delivery
```

File.Deliver This method delivers a message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the Consume method, then the server responds with Deliver methods as and when messages arrive for that consumer.

```
delivery_tag       longlong    #  server-assigned delivery tag
redelivered        bit         #  signal a redelivered message
destination        shortstr    #  destination name
```

File.Browse This method provides a direct access to the messages in a queue or subscription using a synchronous dialogue that is designed for specific types of application where functionality is more important than performance.

```
ticket             short       #  access ticket granted by server
service_type       octet       #  destination service type
destination        shortstr    #  destination name
subscription       shortstr    #  subscription name
no_local           bit         #  do not receive own messages
auto_ack           bit         #  no acknowledgement needed
```

File.Browse Ok This method delivers a message to the client following a browse method. A browsed message will need to be acknowkedged, unless the auto-ack option was set to one.

```
delivery_tag       longlong    #  server-assigned delivery tag
redelivered        bit         #  signal a redelivered message
destination        shortstr    #  destination name
message_count      long        #  number of messages pending
```

File.Browse Empty This method tells the client that the queue or subscription has no messages available for the client. This method has no fields apart from the standard method header.

File.Ack This method acknowledges one or more messages delivered via the Deliver or Browse-Ok methods. The client can ask to confirm a single message or a set of messages up to and including a specific message.

```
delivery_tag       longlong    #  server-assigned delivery tag
multiple           bit         #  acknowledge multiple messages
```

File.Reject This method allows a client to reject a message. It can be used to cancel large incoming messages, or return unprocessable messages to their original queue or subscription.

```
delivery_tag       longlong    #  server-assigned delivery tag
requeue            bit         #  requeue the message
```

3.1.8.10 The Stream Class

The stream class provides methods that support multimedia streaming. The stream class uses the following semantics: one message is one packet of data; delivery is unacknowleged and unreliable; the consumer can specify quality of service parameters that the server can try to adhere to; lower-priority messages may be discarded in favour of high priority messages.

This is the formal grammar for the class:

```
file              = C:CONSUME S:CONSUME_OK
                  / C:CANCEL
                  / C:PUBLISH content
                  / S:DELIVER content
```

This class contains the following server methods:

- Stream.Consume - start a destination consumer (ID = 1) (sync request)
- Stream.Cancel - end a destination consumer (ID = 3) (async)
- Stream.Publish - publish a message to a destination (ID = 4) (async, carries content)

This class contains the following client methods:

- Stream.Consume Ok - confirm a new consumer (ID = 2) (sync reply for Consume)
- Stream.Deliver - notify the client of a consumer message (ID = 5) (async, carries content)

Stream.Consume This method asks the server to start a "consumer", which is a temporary request for messages from a specific queue or topic subscription. Consumers last as long as the channel they were created on, or until they are cancelled.

```
ticket            short      #  access ticket granted by server
service_type      octet      #  destination service type
destination       shortstr   #  destination name
subscription      shortstr   #  subscription name
prefetch_size     short      #  prefetch window in octets
prefetch_count    short      #  prefetch window in messages
consume_rate      long       #  transfer rate in octets/second
no_local          bit        #  do not receive own messages
exclusive         bit        #  request exclusive access
```

Stream.Consume Ok This method provides the client with a consumer tag which it may use in methods that work with the consumer.

```
consumer_tag      short      #  server-assigned consumer tag
```

Stream.Cancel This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer.

```
consumer_tag      short      #  server-assigned consumer tag
```

Stream.Publish This method publishes a message to a specific destination. The message will be saved to the destination and distributed to any consumers when the transaction is committed.

```
ticket            short      #  access ticket granted by server
service_type      octet      #  destination service type
destination       shortstr   #  destination name
immediate         bit        #  assert immediate delivery
```

Stream.Deliver This method delivers a message to the client, via a consumer. In the asynchronous message delivery model, the client starts a consumer using the Consume method, then the server responds with Deliver methods as and when messages arrive for that consumer.

```
destination       shortstr   #  destination name
immediate         bit        #  assert immediate delivery
```

3.1.8.11 The Tx Class

Standard transactions provide so-called "1.5 phase commit". We can ensure that work is never lost, but there is a chance of confirmations being lost, so that messages may be resent. Applications that use standard transactions must be able to detect and ignore duplicate messages.

This is the formal grammar for the class:

```
tx                  = C:COMMIT S:COMMIT_OK
                    / C:ABORT S:ABORT_OK
```

This class contains the following server methods:

- Tx.Commit - commit the current transaction (ID = 1) (sync request)
- Tx.Abort - abandon the current transaction (ID = 3) (sync request)

This class contains the following client methods:

- Tx.Commit Ok - confirm a successful commit (ID = 2) (sync reply for Commit)
- Tx.Abort Ok - confirm a successful abort (ID = 4) (sync reply for Abort)

Tx.Commit This method commits all messages published and acknowledged in the current transaction. A new transaction starts immediately after a commit. This method has no fields apart from the standard method header.

Tx.Commit Ok This method confirms to the client that the commit succeeded. Note that if a commit fails, the server raises a channel exception. This method has no fields apart from the standard method header.

Tx.Abort This method abandons all messages published and acknowledged in the current transaction. A new transaction starts immediately after a rollback. This method has no fields apart from the standard method header.

Tx.Abort Ok This method confirms to the client that the abort succeeded. Note that if an abort fails, the server raises a channel exception. This method has no fields apart from the standard method header.

3.1.8.12 The Dtx Class

Distributed transactions provide so-called "2-phase commit". This is slower and more complex than standard transactions but provides more assurance that messages will be delivered exactly once. The AMQP/Fast distributed transaction model supports the X-Open XA architecture and other distributed transaction implementations. The Dtx class assumes that the server has a private communications channel (not AMQP/Fast) to a distributed transaction coordinator.

This is the formal grammar for the class:

```
dtx                 = C:START S:START_OK
```

This class contains the following server methods:

- Dtx.Start - start a new distributed transaction (ID = 1) (sync request)

This class contains the following client methods:

- Dtx.Start Ok - confirm the start of a new distributed transaction (ID = 2) (sync reply for Start)

Dtx.Start This method starts a new distributed transaction. This must be the first method on a new channel that uses the distributed transaction mode, before any methods that publish or consume messages.

```
      dtx_identifier     shortstr    #  distributed transaction identifier
```

Dtx.Start Ok This method confirms to the client that the transaction started. Note that if a start fails, the server raises a channel exception. This method has no fields apart from the standard method header.

3.1.8.13 The Test Class

The test class provides methods for a peer to test the basic operational correctness of another peer. The test methods are intended to ensure that all peers respect at least the basic elements of the protocol, such as frame and content organisation and field types. We assume that a specially-designed peer, a ”monitor client” would perform such tests.

This is the formal grammar for the class:

```
      test               = C:INTEGER S:INTEGER_OK
                         / S:INTEGER C:INTEGER_OK
                         / C:STRING S:STRING_OK
                         / S:STRING C:STRING_OK
                         / C:TABLE S:TABLE_OK
                         / S:TABLE C:TABLE_OK
                         / C:CONTENT S:CONTENT_OK
                         / S:CONTENT C:CONTENT_OK
```

This class contains the following server methods:

- Test.Integer - test integer handling (ID = 1) (sync request)
- Test.Integer Ok - report integer test result (ID = 2) (sync reply for Integer)
- Test.String - test string handling (ID = 3) (sync request)
- Test.String Ok - report string test result (ID = 4) (sync reply for String)
- Test.Table - test field table handling (ID = 5) (sync request)
- Test.Table Ok - report table test result (ID = 6) (sync reply for Table)
- Test.Content - test content handling (ID = 7) (sync request, carries content)
- Test.Content Ok - report content test result (ID = 8) (sync reply for Content, carries content)

This class contains the following client methods:

- Test.Integer - test integer handling (ID = 1) (sync request)
- Test.Integer Ok - report integer test result (ID = 2) (sync reply for Integer)
- Test.String - test string handling (ID = 3) (sync request)
- Test.String Ok - report string test result (ID = 4) (sync reply for String)
- Test.Table - test field table handling (ID = 5) (sync request)
- Test.Table Ok - report table test result (ID = 6) (sync reply for Table)
- Test.Content - test content handling (ID = 7) (sync request, carries content)
- Test.Content Ok - report content test result (ID = 8) (sync reply for Content, carries content)

Test.Integer This method tests the peer's capability to correctly marshal integer data.

```
      integer_1          octet       #  octet test value
      integer_2          short       #  short test value
      integer_3          long        #  long test value
      integer_4          longlong    #  long-long test value
      operation          octet       #  operation to test
```

Test.Integer Ok This method reports the result of an Integer method.

```
result            longlong    #  result value
```

Test.String This method tests the peer's capability to correctly marshal string data.

```
string_1          shortstr    #  short string test value
string_2          longstr     #  long string test value
operation         octet       #  operation to test
```

Test.String Ok This method reports the result of a String method.

```
result            longstr     #  result value
```

Test.Table This method tests the peer's capability to correctly marshal field table data.

```
table             table       #  field table of test values
integer_op        octet       #  operation to test on integers
string_op         octet       #  operation to test on strings
```

Test.Table Ok This method reports the result of a Table method.

```
integer_result    longlong    #  integer result value
string_result     longstr     #  string result value
```

Test.Content This method tests the peer's capability to correctly marshal content. This method has no fields apart from the standard method header.

Test.Content Ok This method reports the result of a Content method. It contains the content checksum and echoes the original content as provided.

```
content_checksum  long        #  content hash
```

3.1.8.14 Explanatory Notes

Methods are numbered locally per class and there is no attempt to keep a consistent numbering for similar methods in different classes, nor to distinguish client or server methods through special numbering schemes. Method numbering has no significance beyond uniquely identifying the method.

The grammars use this notation:

- 'S:' indicates data or a method sent from the server to the client.
- 'C:' indicates data or a method sent from the client to the server.
- +term or +(...) expression means '1 or more instances'.
- *term or *(...) expression means 'zero or more instances'.
- Methods are indicated by uppercase names, e.g. OPEN.

3.1.9 Queueing and Routing Mechanisms

3.1.9.1 Goals and Principles

The AMQP queueing and routing mechanisms define how content is moved between an AMQP server and its clients. Our goal is to define a single coherent queueing and routing model that can support a range of different highly-functional content classes.

Our design is based on these elements:

- Virtual hosts: an independent collection of destinations.
- Queue destinations: a queue distributes messages between consumers.
- Topic destinations: a topic multiplies messages across subscriptions.

- Subscriptions: a subscription distributes messages between consumers.
- Selection and routing mechanisms: allowing consumers and subscriptions to be specific about which messages they receive.
- Message persistence: defining the level of assurance that a message will be successfully delivered despite crashes and failures.
- Message priorities: allowing messages to be delivered out-of-sequence according to their importance.
- Destination browsing: allowing a client to read messages in a simple synchronous manner.
- Acknowledgements: allowing a client to tell the server when it has successfully processed a message.
- Window-based flow control: allowing a client to control how many messages it receives
- Transactions: allowing a client to group multiple messages and acknowledgements into a single unit of work.

In general all these mechanisms apply to all content classes. Certain content classes may elect to use a simpler model for performance reasons. This is specifically the case for streamed content, which does not support persistence, browsing, acknowledgements, or transactions.

3.1.9.2 Virtual Hosts

AMQP/Fast has explicit support for virtual hosts. The client chooses a virtual host as the last stage in negotiating the connection. A virtual host has its own set of destinations and access controls.

3.1.9.3 Queue Destinations

A queue is a named object that acts as an asynchronous buffer for messages. Queues may be held in memory, on disk, or some combination. Each virtual host provides a namespace for queues. Queues hold messages and distribute them between one or more clients (consumers). A message published to a queue is never sent to more than one client unless it is is being resent after a failure or rejection.

Note that mixed content on a queue is generally sent asynchronously using the appropriate NOTIFY methods, to those clients that have asked for it. The BROWSE method allows clients to selectively receive single JMS or File messages but not Stream messages.

3.1.9.4 Topic Destinations

Topics are named objects that act as multipliers. Topics do not hold messages: rather they copy messages to subscriptions. Each virtual host provides a namespace for topics. A message published to a topic is duplicated to each subscription that has asked for it (using one of the selection mechanisms explained below).

3.1.9.5 Subscriptions

Subscriptions are named or unnamed objects that resemble queues and may be implemented using the same internal mechanisms in the server. Each virtual host provides a namespace for subscriptions. Subscriptions may be held in memory, on disk, or some combination. Subscriptions can have a single consumer (the normal case) but also multiple consumers, in which case their messages are distributed between them in the same way as messages published to a queue.

3.1.9.6 Selection and Routing

AMQP/Fast provides these mechanisms for a client to select messages, and for a server to route them:

1. By destination name.
2. By destination pattern, for topics only.

3. By message properties, for queues and topics. This mechanism is called a ”selector”.
4. By implementation-dependent functionality embedded in the server.

3.1.9.7 Message Persistence

A persistent message is held securely on disk and guaranteed to be delivered even if there is a serious network failure, server crash, overflow etc.

Messages may be persistent or not, depending on several criteria:

1. In the JMS content class, messages are individually marked as ”persistent” or not.
2. In the File content class, messages are always persistent.
3. In the Stream content class, messages are never persistent.

3.1.9.8 Message Priorities

A high priority message is sent ahead of lower priority messages waiting in the same queue or subscription.

3.1.9.9 Acknowledgements

When the client finishes processing a message it tells the server via an acknowledgement. AMQP/Fast defines different acknowledgement models for different levels of performance and reliability:

- Acknowledgement by the application after it has processed a message.
- Acknowledgement by the client layer when it has received a message and before it passes it to the application.
- Acknowledgement by the server when it has sent a message to the client.

The first two models use the same protocol mechanics. The difference is when the client API sends the acknowledgement. The last model uses a specific protocol option, in the CONSUME method for those classes that support acknowledgement.

3.1.9.10 Flow Control

AMQP/Fast flow control is based on these concepts:

1. Allowing the client to specify a prefetch - how many messages it will receive without acknowledgement. We call this ”window-based flow control”.
2. Allowing one peer to halt and restart a flow of messages coming from the other peer, using the Channel FLOW method, which we call ”channel flow control”.

Since the PUBLISH method is not acknowledged, there is no window-based flow control for publishing and the recommended way to pause a publisher is to use channel flow control.

3.1.9.11 Transactions

AMQP/Fast supports three kinds of transactions, each kind progressively slower, more complex, and more reliable:

1. Automatic transactions, in which every method is wrapped in a transaction. Automatic transactions provide no way to create units of work. Every message and acknowledgement is processed as a stand-alone transaction.

2. Simple transactions, which cover units of work on a per-channel basis. Transactions cover messages published, and messages received and acknowledged. There is no ”start transaction” method: a transaction starts when the channel is opened and after every commit or rollback. If the channel is closed without a commit, all pending work is rolled-back. Nested transactions are not allowed.
3. Distributed transactions, which include XA-compatible transactions.

The transaction type is chosen on a per-channel basis.

3.1.10 Error Handling

3.1.10.1 Goals and Principles

Error handling is a critical aspect of any protocol. First, we need a clear statement of what situations can provoke an error. Secondly, we need a clear way of reporting errors. Lastly we need unambiguous error handling that leaves both peers in a clear state.

The AMQP/Fast error handling model is based on an answer to each of these questions:

1. Use an exception-based model to define protocol correctness.
2. Use existing standards for error reporting.
3. Use a hand-shaked close to handle errors.

3.1.10.2 Existing Standards

The standard for error handling (defined semi-independently in several IETF RFCs) is the 3-digit reply code. This format has evolved into a fine-grained tool for communicating success or failure. It is also well-structured for expansion as a protocol gets more mature.

The current AMQP reply codes are standard to all protocols in the AMQP family and are defined in AMQ RFC 011.

The reply code is constructed as follows:

- The first digit - the completion indicator - reports whether the request succeeded or not.
- The second digit - the category indicator - provides more information on failures.
- The third digit - the instance indicator - distinguishes different situations with the same completion/category.

The completion indicator has one of these values:

```
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.
```

The category indicator has one of these values:

```
0 = error in syntax.
1 = the reply provides general information.
2 = problem with session or connection.
3 = problem with security.
4 = application-specific.
```

The instance indicator is 0 to 9 as needed to distinguish different situations.

3.1.10.3 The Assertion/Exception Model

AMQP/Fast uses an assertion/exception model that has these goals:

- identify and document all protocol preconditions (”assertions”).

- define the exception level caused by any assertion failure.
- define a formal procedure for handling such exceptions.

AMQP/Fast defines two exception levels:

1. Channel exceptions. These close a single virtual connection. A channel exception is raised when a peer cannot complete some request because of transient or configuration errors.
2. Connection exceptions. These close the socket connection. A connection exception is raised when a peer detects a syntax error, badly-formed frame, or other indicator that the other peer is not conformant with AMQP/Fast.

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

3.1.10.4 Hand-shaked Closure

Closing a connection for any reason - normal or exceptional - must be done carefully. Abrupt closure is not always detected rapidly, and in the case of errors, it means that error responses can be lost. 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.

A peer can close a channel or connection at any time for internal reasons, or as a reaction to an error. It sends a Close method to the other party. The receiving peer must respond to a Close with a Close-Ok method. The closing peer reads methods back until it gets a Close-Ok, at which point it closes the connection and frees resources.

3.1.11 The JMS Operational Model

3.1.11.1 Goals and Principles

JMS is a standard API for messaging middleware brokers. The AMQP/Fast JMS implementation is compatible with providers that conform to the SUN JMS specifications (within the bounds of JMS standardisation, which does not guarantee interoperability). JMS messages can be published and consumed using a set of methods that map cleanly to the JMS API, including:

- Publishing a message to a destination.
- Creating a consumer for a destination.
- Browsing a destination for messages.

3.1.11.2 JMS Content Properties

These are the properties defined for JMS content:

- ContentType (tiny string)
- ContentEncoding (tiny string)
- DeliveryMode (octet)
- Priority (octet)
- CorrelationID (tiny string)
- ReplyTo (tiny string)
- Destination (tiny string)
- Expiration (tiny string)
- MessageID (tiny string)

- Timestamp (tiny string)
- Type (tiny string)
- UserID (tiny string)
- AppID (tiny string)
- Headers (field table)

3.1.12 The File Transfer Operational Model

3.1.12.1 Goals and Principles

The file transfer operational model is designed for enterprise-wide file transfer based on the AMQP/Fast queueing and routing models. For instance, routing files via the publish and subscribe functionality of topics and subscriptions.

File content is specifically different from JMS content in that:

- Files are always persistent.
- File transfer is restartable: if a file has been partially transferred when a connection is broken, the sender can resend just the remainder.

File transfers always happen in two steps:

1. The sender "stages" the file into a temporary area provided by the recipient.
2. The sender notifies the recipient of the file, using Publish (a client to a server), or Deliver (a server to a client).

A File message is persistent and optimised for restartable transfers across possibly unreliable network connections.

3.1.12.2 File Content Properties

These are the properties defined for File content:

- ContentType (tiny string)
- ContentEncoding (tiny string)
- Priority (octet)
- ReplyTo (tiny string)
- Destination (tiny string)
- MessageID (tiny string)
- FileName (tiny string)
- Timestamp (tiny string)
- Headers (field table)

3.1.13 The Stream Operational Model

3.1.13.1 Goals and Principles

Streaming is intended for multimedia applications: video, music, etc. A stream consists of an unterminated series of messages, each message containing a fragment of streamed data.

The AMQP/Fast streaming model separates the compression technology (codec) from the streaming technology. The streaming model is compatible with any codec that allows data to be fragmented.

Streamed messages use the full AMQP/Fast queueing and routing mechanisms with some simplifications:

1. Streamed content is non-persistent though the server MAY hold it on disk if needed.
2. Streamed data sent to a queue cannot be shared between multiple consumers (each would receive partial streams).
3. Streamed data sent to a subscription via a topic cannot be shared between multiple consumers (each would receive partial streams).

A stream producer can use topics to distribute streams to multiple consumers at once. Topics can be used - e.g. - to provide different qualities of the same stream.

A stream consumer can specify:

- Prefetch size in octets and/or messages.
- Desired rate in octets per second.

3.1.13.2 Stream Content Properties

These are the properties defined for Stream content:

- ContentType (tiny string)
- ContentEncoding (tiny string)
- Priority (octet)
- Destination (tiny string)
- Timestamp (tiny string)
- Headers (field table)

3.1.14 Security

3.1.14.1 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.

3.1.14.2 Buffer Overflows

All data is length-specified so that applications can allocate memory in advance and avoid deadlocks. Length-specified strings protect against buffer-overflow attacks.

3.1.14.3 Denial of Service Attacks

AMQP/Fast handles 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, or failed attempts to authenticate. The server implementation should log all such exceptions and flag or block clients provoking multiple failures.

3.1.15 Miscellaneous Topics

3.1.15.1 Performance

To be completed.

3.1.15.2 Message Templating

To be completed.

3.1.15.3 Field Dictionaries

To be completed.

3.1.15.4 Message Selectors

To be completed.

3.1.15.5 Topic Selectors

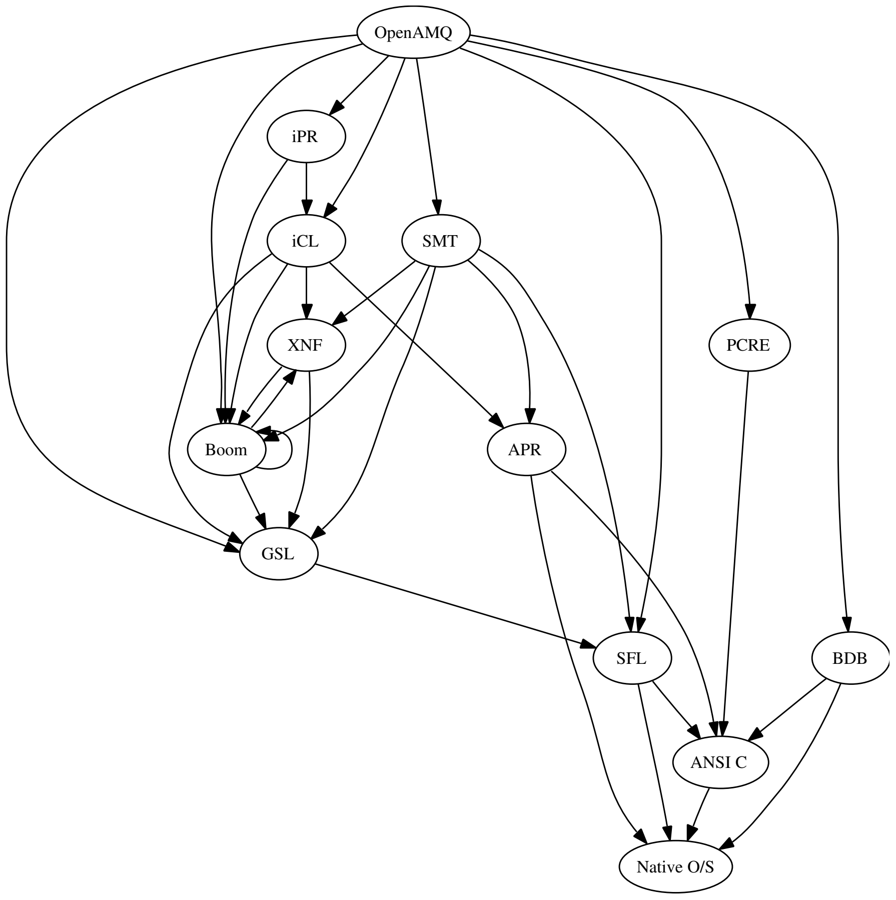
To be completed.

3.1.15.6 Destination Templating

To be completed.

4 The Toolset

In which we identify and name the giants upon who's shoulders we are standing.



4.1 Standard Libraries

4.1.1 Goals and Principles

The use of standard libraries is a critical choice in the architecture of any product. The wrong choices can create an unstable product that is difficult to build and install and ultimately unsatisfying for its users. Underuse of standard libraries creates larger code bases with no real benefit.

We work with these principles when choosing and writing standard libraries:

1. Our products must be entirely self-supporting, able to build on a "raw" box with only the basic tools (compiler and linker).
2. Any external libraries we use must use a BSD-style or public-domain license so that we do not risk GPL "contamination". (This is ironic because we use the GPL for our own iMatix libraries except in the context of OpenAMQ.)
3. Any external libraries we use must be entirely stable and portable to all our target systems, which means Linux, Solaris and other modern Unixes, and Win32.

4. Any non-portable code must be isolated in a portability library module. That is, our application code does not use conditional compilations for portability.
5. If we cannot find external libraries that do what we need, we write our own libraries.

The above principles are well-tested. We have successfully used them to write high-quality portable servers that run on many different OSes.

4.1.2 Apache Portable Runtime (APR)

APR (apr.apache.org) provides a fairly rich set of functions for encapsulating non-portable system functionality such as socket access, threading, and so on. We use APR moderately but increasingly as we develop OpenAMQ.

APR is solid and clearly designed to support a communications product such as OpenAMQ. It is immature in several areas: weak documentation, incomplete APIs, and some portability issues, but overall we feel it's worth using.

4.1.3 Sleepycat Berkeley Database (BDB)

BDB (sleepcat.com) is a mature ISAM system. Our experience with it is somewhat contradictory. On the one hand, the product is well documented, widely used in other projects, and very stable in operation. On the other hand, it is literally painful to program. If you use the wrong sequence of calls, it will simply abort, without any explanation. Writing BDB code is often a matter of trial and error. All our calls to BDB are in generated classes, so the pain of writing BDB code is one-time only.

We use BDB for message persistence, and for object persistence.

We are not entirely satisfied with BDB's performance: in order to get a stable application, transactions must be used for all operations (or we get those famous aborts). Thus even database reads are transactional, and the transaction logs tend to grow. We will probably write our own persistence layer at some stage.

4.1.4 Hazel's Perl Compatible Regular Expressions (PCRE)

The PCRE library is a small and useful library that we use for parsing and matching. For instance in OpenAMQ this library does the parsing and matching of topic names.

4.1.5 iMatix Standard Function Library (SFL)

SFL provides a similar library to APR. It is more mature, and ported to many platforms. SFL is considered "legacy", in the sense that we are not adding new functionality to it. However we hold SFL as a backup for portability to systems that APR may not yet support.

4.1.6 iMatix Portable Runtime (iPR)

iPR is our modern replacement for those functions in SFL that we want to use, and which are not provided by APR. iPR is written using the iCL framework (see below).

4.2 Languages

4.2.1 Goals and Principles

Our choice of languages is driven by these principles:

1. Our languages must be portable and mature.

2. All production software must be written in C.

The choice of C is somewhat driven by conservative ideas about writing code that is very fast, and predictably so. A C operator is entirely predictable. A C++ operator may be overloaded in ways that are not obvious.

The choice of C is also comforted by the observation that the language is not a factor in writing good, well-abstracted code. i.e. the choice of C, C++, ObjectiveC, etc. matters much less than the choice of tools. It is, we think, a naive notion to expect the language to solve the abstraction problems we face unless we can scale the language considerably above the level most compilers provide. LISP is an interesting language, being scalable in the way we like. But there are few LISP programmers. Open source must be accessible. We therefore choose C and scale the language using code generation.

4.2.2 ANSI C

We use ANSI 1989 C with a small set of necessary extensions that are portable to our target systems (inlined functions, 64 bit integers). On Linux and Unix systems we use gcc.

4.2.3 iMatix Generator Scripting Language (GSL)

We use the GSL code generator construction language to build the many code generators that we use. GSL is the result of some 20 years of research into code generation, and highly tuned to the job. In 1991 we wrote a code generator - Libero - that turns finite-state diagrams into code, a very useful tool for writing servers, parsers, and such. The code generator took several months to write (not including extensive documentation, a Windows GUI and so on). In 2005, we can write a Libero-equivalent code generator in GSL in a matter of a few hours.

In most cases we actually generate the code generators - parsers, inheritors, and command-line wrappers - using the XNF framework (see below).

4.2.4 Perl

We use Perl in some parts of the process where GSL is not flexible enough, mainly for document preparation. Perl is widely available and very useful, but tends to produce unmaintainable code with exotic and complex package dependencies, so we do not rely on it very heavily.

4.3 Code Generation Frameworks

4.3.1 Theological Principles

iMatix has a long tradition of building and using tools to support our software development process. Our principle technology is code generation, which has evolved over the years as follows:

1. Hard-coded code generators that produce specific outputs given specific input files. (ETK, 1985-1991). Clues: the code generator uses "print" statements or templates with fixed structures.
2. Template-based code generators that produce arbitrary outputs from specific input files (Libero, 1991-present). Clues: the code generator has some type of templating language.
3. Template-based code generators that produce arbitrary outputs from arbitrary structured input in XML or a similar language (GSL, 1995-present).
4. The definition of standard abstraction models and techniques to turn this "arbitrary structured input" into something resembling a programming language (Aqua, Changeflow, 2000).
5. The definition of meta-languages that allow these XML languages to be formally defined and validated (Boom, 2002).

6. The automatic generation of GSL code generators to implement such XML languages (XNF, 2004).

Other projects use code generation too. For instance Samba/4 is 57% code generated according to its author. The Samba/4 code generator, pidl, generates RPC marshalling code, test cases, documentation, and other outputs from IDL (interface definition languages) specifications. While pidl is a great achievement, it basically represent early code generation technology.

Our use of code generation has a profound impact on the quality and cost of the software we make. While code generation is starting to become a standard part of a professional's toolbox, it is generally limited to IDLs, and class structures (as produced by analysis models such as UML). We use code generation to implement "fat languages", a kind of macro programming that produces - in our experience - superb results. In the words of the author of Samba/4, "code generation is addictive. I wish we'd started ten years ago."

The advantages of a mature code generation technology are:

- Rapid and cheap improvement of the code generation templates, so that the generated code can be incrementally improved until it is as good as hand-written code.
- Ability to create new models and macro languages cheaply, so that we can solve problem domains rapidly.
- Use of simple XML for all models, including XML grammars, so that we can use XML validation and meta-validation on all inputs.
- Ability to produce any text output: programming languages, documentation, test cases, etc., so that we can extend the code generation process beyond pure source code.
- Ability to build large-scale code generation frameworks out of smaller individual code generators.

We can measure the power ratio of a code generation framework by comparing the cost of writing the abstract model against the cost of writing the code by hand. Not all generated code is worth having, of course, but then not all hand-written code is worth having either.

The power comes from abstraction. i.e. defining standard models and then deriving specific instances from these. All our modern code generators are heavily based on the concept of inheritance.

Here is the typical leverage provided by the code generation frameworks described in this section:

- XNF, generating GSL parsers and generators - 500%
- SMT, generating state machine engines - 750%
- iCL, generating class hierarchies - 750%
- AMQP/Fast frames generator, generating marshalling code - 1,500%
- Boom, generating makefiles and build scripts - 10,000%

4.3.2 iMatix Class Library (iCL)

iCL is a software development methodology and toolkit aimed at building very large and very high-quality applications in ANSI C. We use C in infrastructure projects because it is portable, fast and operationally stable. However, C lacks modern facilities such as:

1. Inheritance (where functionality can be added by extending and building on existing components)
2. Templating (where variations on a standard model can be produced cheaply)
3. API standardisation (where the structure of APIs is enforced by the language).
4. Literate coding (where documentation can be written together with the code)
5. Logical modelling (where we define abstract models such as finite state machines directly in the code)

These can be done manually, and usually are, but that is expensive and demanding.

These facilities are helpful because they allow us to work faster and with better results. We can make larger and more ambitious designs with less risk of losing control over them. iCL adds these facilities to C by wrapping it into a higher level class library which is self-documenting, simple, extensible and language independent. iCL adds a stage to the deployment process, but unlike interpreted paradigms, portability and efficiency are guaranteed as an atomic quality that follows as a natural consequence of the target language, ANSI C. Our solution aims to leverage skills which are normally part of the essential training of any IT professional (C, XML).

We write our programs as iCL classes, using a simple XML language. The XML language (called "iCL") is designed to be easy to read and write by hand - no special editors are needed. An iCL class consists mainly of a set of properties and a set of methods. The properties define an object's state. The methods define an object's functionality.

iCL classes compile into C code, which we then compile and link as usual. In a best-case scenario, we can see up to a 10-to-1 ratio between the final C code and the hand-written iCL classes, but coding effort compression is typically non-negligible. The generated C code is perhaps 20-30% larger than a comparable hand-written application, but this code is typically flatter - more inlined - and produces faster programs.

iCL does not attempt to create a full OO model. It aims to remove much of the administrative burden from writing large-scale C applications, to produce high-quality code, and to ensure that certain problems - such as memory management - can be totally abstracted and thus solved "correctly" once and for all.

4.3.3 iMatix Simple Multithreading Kernel (SMT)

SMT/4 is the latest version of this kernel, which has driven our servers for about a decade. SMT/4 is built as an iCL application with a custom code generator. It provides:

- A finite-state model for writing "agents", which implement protocol handlers and other FSM-defined components.
- Functions for starting and stopping threads of control in an agent.
- Functions for communicating between threads and between non-SMT code and SMT threads, using event queues,
- Asynchronous socket i/o functions.
- Private context areas for each thread.
- Standard agents for logging, timer alarms, error handling, etc.
- The ability to package arbitrary agents together into applications.
- Overall management functions for starting and stopping SMT applications.

The OpenAMQ server, and the kernel clients, are written as SMT agents. SMT/4 is quite a radical improvement over SMT/3, which used the Libero code generation tool. SMT/4 is full XML, and generates extremely flat and fast finite-state machines (avoiding loops for most cases). An SMT/4 application runs about twice as fast as an SMT/3 application.

4.3.4 iMatix XML Normal Form (XNF)

XNF is a technology that we started building in 1998 to support very large-scale code generation frameworks. It is similar to a compiler construction toolkit such as Yacc, but working within the XML/GSL domain.

An XNF specification describes a specific XML language and a set of rules about the code generators that use it. The XNF compiler turns these specifications into complete working code generators. XNF embodies our "best practise" for code generators. For small ad-hoc code generators it is rather a complex tool. For major code generators such as iCL, it automates much of the most complex work at low cost.

Our standard code generation model compiles a high-level XML program into native code as follows:

- Load the XML tree and perform basic XML syntax validation.
- Perform XML content validation and set default values where possible.
- Resolve inheritance rules between different parts of the XML tree.
- Run one or more target scripts that produce the final output.

XNF solves these parts of the process:

1. Defining a grammar for the XML language so that it can be validated. XNF generates a preprocessing parser in GSL which validates the XML, provides default values for attributes, etc. The XNF grammar can also be used - in theory - to drive XML-aware IDEs such as Eclipse.
2. Documenting the language. An XNF specification contains textual explanations for every item, every attribute, and XNF produces gurudoc output containing presentable documentation.
3. Resolving inheritance. XML languages are hierarchical and we use inheritance as a way of eliminating redundancy. For instance in a complex finite-state machine, several states could inherit their structure from an abstract parent state. Or in a class structure, methods can inherit their structure from a method template. XNF defines a set of inheritance mechanisms which may be applied to each tag in the language, and it generates a GSL program that executes the inheritance rules on a validated XML tree.
4. Wrapping the whole process in a command-line GSL script that loads and validates the XML, resolves inheritance, and runs the target scripts to produce final output.

XNF is thus a code generator for building code generators. iCL, SMT, boom, and XNF itself are all built using XNF.

This is a summary syntax for XNF:

```
<xnf name version [before] [after] [abstract] [script] [copyright]
   [role]>
   <option name value/>
   <inherit name>
      <option .../>
   </inherit>
   <include filename/>
   <produce filename type/>
   <entity name [abstract] [template] [tag] [cdata] [key] [unique]
        [inherit] [sequence] [disconnect] [export]>
      <option .../>
      <inherit .../>
      <allow entity [occurs] [inherit] [sequence] [export]/>
      <rule [phase] [when]/>
      <attr name [required] [default] [inherit] [phase]>
         <restrict value/>
      </attr>
      <link from [entity] [field] [required] [disconnect]>
         <rule [phase] [when]/>
      </link>
   </entity>
   <errorhandler/>
   <rule [phase] [when]/>
</xnf>
```

4.3.5 Ad-Hoc Frameworks

OpenAMQ has a number of ad-hoc code generation frameworks that are not large enough to be worth defining with XNF:

- The AMQP/Fast frames generators, in versions that produce C, Java, and Perl code respectively.
- The openamq.org web site generator.

4.4 Documentation

4.4.1 iMatix Gurudoc

Gurudoc is our standard way of producing documentation and web sites. It has the particular advantage of being extremely lightweight and fast to use, while producing high-quality output in different formats.

A gurudoc document looks like a neatly-formatted text file. The gurudoc parser recognises simple layout rules in the text and uses those to produce a formally structured document that can be turned into higher-quality outputs. This document is written using gurudoc.

Various gurudoc templates produce varieties of HTML (with frames, without frames), LaTeX (allowing PDF generation), simple text, OpenOffice.org, RTF, etc.

4.4.2 LaTeX

LaTeX is used to produce PDFs. It is a huge system and does not build on all our workstations. But it produces very elegant PDFs.

4.4.3 AT&T Graphviz

Graphviz turns simple digraphs into elegant images. We use Graphviz to produce the state and class diagrams used in this document. Graphviz is integrated into the gurudoc toolchain.

4.4.4 Wiki

The openamq.org web site provides a wiki whiteboard for developers. We use the Oddmuse wiki which is a simple and functional implementation. We have modified the code to support gurudoc markup for wiki pages.

4.5 Project Management

4.5.1 iMatix Boom

Boom is a portable command-line toolkit that automates the build process for software projects. In simple terms, boom takes a project description and turns this into various platform-dependent scripts and makefiles that do the hard work of turning source code into executable files.

The short answer to the standard question "why use boom instead of make or autoconf" is that make files are not portable, autoconf is complex, and the build process requires more than these tools provide.

The longer answer is that boom projects are particularly cheap to make maintain since files' properties are abstracted (using inheritance, as usual) and relationships between files are extracted automatically. Further, boom produces packages (source & binary) at no extra cost.

Boom produces make and configure scripts to give people a familiar user interface. Here is a typical project definition (for the OpenAMQ documentation project, which includes this document):

```
<?xml?>
<pdl name = "OpenAMQ Documentation" version = "0.8c3">
<include filename = "prelude.pdl" />
<file name = "concepts.txt"          class = "gurudoc text" />
<file name = "gdstyle.css"           class = "web resource" />
<file name = "mainlogo.jpg"          class = "web resource" />
<file name = "amq_server_agent.dot"  class = "digraph" />
<file name = "amq_sclient_agent.dot" class = "digraph" />
<file name = "amq_aclient_agent.dot" class = "digraph" />
<file name = "server_classes.dot"    class = "digraph" />
<class name = "digraph" inherit = "private resource">
    <derive extension = ".png"  class = "generated" />
    <generate target = "unix">
        <execute command = "builddot" />
    </generate>
</class>
</pdl>
```

Here is an edited fragment of the generated code (this implements the 'distsrc' command that builds a source package):

```
rm -f _package.lst
echo concepts.txt>>_package.lst
echo gdstyle.css>>_package.lst
echo mainlogo.jpg>>_package.lst
echo amq_server_agent.dot>>_package.lst
echo amq_sclient_agent.dot>>_package.lst
echo amq_aclient_agent.dot>>_package.lst
echo server_classes.dot>>_package.lst
echo prelude.pdl>>_package.lst
echo license.gpl>>_package.lst
echo project.pdl>>_package.lst
echo readme.txt>>_package.lst
for file in `echo concepts*.html`; do
    echo $file>>_package.lst
done
echo amq_server_agent.png>>_package.lst
echo amq_sclient_agent.png>>_package.lst
echo amq_aclient_agent.png>>_package.lst
echo server_classes.png >>_package.lst
echo stamp_generate>>_package.lst
echo configure>>_package.lst
echo boomconf>>_package.lst
echo Makefile_dev.unix>>_package.lst
echo Makefile_src.unix>>_package.lst
echo boomake>>_package.lst
echo configure.bat>>_package.lst
echo boomconf.bat>>_package.lst
echo Makefile_dev.win32>>_package.lst
echo Makefile_src.win32>>_package.lst
echo boomake.bat>>_package.lst
echo "Building OpenAMQ_Documentation-0.8c3-src.tar.gz..."
zip  -rq _package.zip -@<_package.lst
unzip -q _package.zip -d OpenAMQ_Documentation-0.8c3
rm -f OpenAMQ_Documentation-0.8c3-src.tar.gz
tar -czf  OpenAMQ_Documentation-0.8c3-src.tar.gz OpenAMQ_Documentation-0.8c3
rm -f OpenAMQ_Documentation-0.8c3-src.zip
echo "Building OpenAMQ_Documentation-0.8c3-src.zip..."
zip -lrmq OpenAMQ_Documentation-0.8c3-src.zip OpenAMQ_Documentation-0.8c3
rm _package.zip
rm _package.lst
```

In typical projects boom produces about 100 lines of build code for one line of project definition. That is good leverage.