Concepts and Vision
A Background to the OpenAMQ Project

version 1.0

Pieter Hintjens <ph@imatix.com>

Revised: 2006/05/21

Contents

1 Cover

1.1 State of this Document

This document is a design white paper.

1.2 Copyright and License

1.3 Abstract

This is a general introduction to the OpenAMQ project, 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.

2 Introduction

2.1 Aim of this Document

This is a general introduction to the OpenAMQ project, 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.

Some of the details in this document are out of date, as it reflects the state of the OpenAMQ project in mid-2005.

2.2 Advanced Message Queues

AMQ ("Advanced Message Queues") was a broad and long-term project with the overall goal of creating commodity middleware for use in large-scale business and financial applications. OpenAMQ was the original reference implementation of the protocol and has become a significant product in itself, and is the focus of this whitepaper. The AMQ project had these specific short term goals:

1. To define an industry-standard wire-level protocol (AMQP).
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).

2.3 Critical Success Factors

We identified these primary success factors:

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

2.4 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.

2.5 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.

2.6 Using Middleware in Real Life

2.6.1 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.

2.6.2 Extreme Scenario - Market Data

Market data is an extreme scenario at the upper-end of what we are aiming for with OpenAMQ. 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.

3 Challenges and Architectures

3.1 Designing a Successful Project

The OpenAMQ 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.

3.2 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 and OpenAMQ - 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.

3.3 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 AMQ protocol, and OpenAMQ 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.

3.4 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.

3.5 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.

3.6 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 AMQP, for several reasons: it uses XML wrapping that we felt was unnecessary and it does not have an actively-developed C implementation.

However, AMQP 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.

3.7 Quality and Standards

3.7.1 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 standards guide for OpenAMQ:

3.7.1.1 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.

3.7.1.2 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}.

3.7.2 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, mat>bi>bits)) {
            subscr = (amq_subscr_t *)>subs>>subs>data [su>data [subscr_nbr];
           >no_local ==>no_local == FALSE
        >cli>       >cl>cli>       >client_id>       >client_id) {
                if (publish)
                    am>xn);
    >queue, messag>xn);
    >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".

3.7.3 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.

3.7.4 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.

3.8 The Server Architecture

3.8.1 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.

3.8.2 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 cores.

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 asynchronous 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).

3.8.3 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 a state machine that accepts network events, and other methods and processes them according to the current state of the connection.

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

```
<state name = "initialise connection">
 <ev<event name = "ok" nextstate = "expect connection respon>
 >
  <actio<action name = "read protocol hea>
    ><action n<action name = "check protocol >
     <action name<action name = "send connection >
     <actio<>
<ac<ev<actio<>
<ac<event<n>
```

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

3.8.4 Server Classes

The server is built as a hierarchy of 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 also very useful.

3.8.5 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.

3.9 Client Architecture

3.9.1 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, will use a similar set of concepts and tools, and provide the application programmer with a similar API, which we call WireAPI.

The client stack consists of a number of layers:

- iMatix SMT provides a multithreading network i/o layer that delivers network traffic to a state machine.
- The protocol agent implements state machines that accept or reject the incoming network frames depending on the state of the connection and channel.
- The session layer provides an external API (WireAPI) for application developers, exposing a set of methods that let the developer work both with AMQP concepts (e.g. 'basic_consume (queue, options)') and with contents.
- The session layer may itself be wrapped in other layers to provide interfaces to scripting languages and other APIs.

3.10 An Open Source Project

This section is a general analysis of the ecology and rationale of an open source project such as OpenAMQ.

3.10.1 Goals and Economics

AMQP was conceived from the start as an open standard, and OpenAMQ as an open source reference implementation for AMQP. This vision is probably the most radical and defining aspect of the entire project.

The AMQP and OpenAMQ projects, funded by a large institution and built by a specialist engineering firm, represent a new way of building software systems. As it grows, OpenAMQ 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 OpenAMQ 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.

3.10.2 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, we have some understanding, or at least strong 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 - most significantly 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/MIT/Apache licenses, because of this effect. We'll come back to this argument.

3.10.3 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.

3.10.4 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?"

3.10.5 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 BSD/MIT model 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 traditional 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 BSD-licensed libraries. Why? Because we can relicense our GPL code for commercial clients, and provide this together with BSD-licensed 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, BSD-style 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. Of course it's possible that Apache would have been much more successful as GPL software; today there are numerous spin off products that used Apache code but "gave nothing back".

It is hard to generalise about the kinds of project that each license attracts but anecdotal evidence suggests that BSD-style 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 BSD-licensed 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, our target uses want the BSD license for clear reasons.

3.10.6 OpenAMQ License Design

Our solution to the apparent paradox noted above is to split the OpenAMQ code into three distinct pieces:

1. The original XML models used as input for the code generation process.
2. The generated C code and small amount of hand-written C code that constitutes the OpenAMQ server.
3. The generated C code and small amount of hand-written C code that constitutes the OpenAMQ client layers (WireAPI).

We license the first and second layers of code under the GPL. Any code that derives from the original XML models is GPL'd unless someone paid for a commercial license. The WireAPI code is BSD-licensed. It lets application developers include it in their work without any GPL impact.

The generated code is clean but not practical for modification; it is object code, the product of machines, and both pedantic and voluminous.

So we are reasonably sure that useful contributions are provided to the GPLed code, while we allow end-users to have BSD-licensed code.

3.11 Escaping the "Version 2 Syndrome"

A product like OpenAMQ 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.

3.11.1 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 OpenAMQ 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.

3.11.2 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 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.

3.11.3 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.

3.12 Prior Art

3.12.1 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.

3.12.2 The SMT Multithreaded Kernel

The SMT multithreaded kernel is based on work done by Pieter Hintjens in 1992-93 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 (AirTours), processing several million travel bookings per year and still in operation in 2006.

3.12.3 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.

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 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.

4.1.4 iMatix Portable Runtime (iPR)

iPR is our modern replacement for those functions in iMatix 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 Model Oriented Programming

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 into "Model Oriented Programming", or MOP, 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.

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 State Machine Threadlets (SMT)

SMT/5 is the latest version of this package, which has driven our servers for about a decade. It provides:

- A finite-state model for writing "agents", which implement protocol handlers and other FSM-defined components.
- Full support for multithreaded applications written using virtual threads.
- 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 WireAPI clients, are written using SMT. SMT/5 is quite a radical improvement over SMT/3, which used the Libero code generation tool. SMT/5 is fully multithreaded, and generates extremely flat and fast finite-state machines (avoiding loops for most cases). An SMT/5 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]>
<op<option name val>
<inher<inheri>
   <op>
   ><op>
   >
<i<<[abst<]<<i<<[abst<]<<enti<<[abstract]<<entity n<[abstract] [te<entity name [abstract] [template] [tag] [cdata] [key] [u
   <] [w<<f>
```

4.3.5 Abstract Server Layer (ASL)

We invented ASL as a modeling language for the AMQ Protocol. Rather than write a new AMQP client and server stack (after the third major design version of AMQP) we decided to write a modeling language that would let us generate the necessary code.

From the AMQP specifications, ASL will generate 100% of the client stack automatically. This requirement drove the design of AMQP to some extent.

4.3.6 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 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, ODT, 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 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<pd<pdl name = "OpenAMQ Documentation" version = "0.<inclu<include filename = "p< =<file < =<file name = "concepts.txt
<class< "digra<   <riv<t<o>  <h>
<class< "digra<   <riv<tensio>  <h>
<class < = "digra<   <riv<tensio>  <h>
<class name = "digra<   <rive extensio>  <h>
<class name = "digraph>
   <rive extensio>  >
```

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 gdsty>>_pac>>_package.lst
echo >>_package.l>>_package.lst
echo am>>_package.lst
ec>>>_package.lst
ec>>_package.>>>>ing OpenA>>_package.>>>>ing OpenAMQ_Docume>>_packa>>>>ing OpenAMQ_Docume>>_package.>>ech>>ing OpenAMQ_Docu
...
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.