This is a redacted text by Pieter Hintjens that was distributed to some AMQP working group members in August 2008, and which covers the main issues with AMQP as seen by iMatix at that time.

Introduction

This article is about AMQP, the Advanced Message Queuing Protocol. AMQP is an attempt by some very large businesses, and some smaller suppliers, to change the rules in the business messaging market. In some ways, they are succeeding. This is a market with two dominant suppliers, IBM and Tibco, and a serious lack of open standards and open-source competitors. It's bad for clients, but also for vendors. Put simply: the lack of real competition in an area of technology means that nothing grows on top. And in technology, all new wealth is created by building layers on top. Sterility is good for no-one: the old money eventually runs out. Look at HTTP, SQL, Linux, and the new wealth that has grown on top of these now basic technologies. Now look at messaging… only in the Java world do we see some growth above messaging. But that's not enough.

Let me explain very briefly how standards work. You might think it's all about making revolutionary new concepts available to the public. In fact, standards are more about stopping innovation than opening the door to it. Well, to be accurate, by freezing innovation at one level, we allow more of it at higher levels.

So standardization always works from the bottom up, from more basic and broadly-used technologies to more sophisticated and narrowly-used ones. Over time the stack of standards gets compressed like seams of sediment, and unused stacks fall away, leaving fewer and fewer basic standards underpinning the whole world of computing. AMQP is quite unusual in that it starts low on the stack, building on protocols like TCP rather than HTTP. It is also unusually ambitious in going from very low level (data on the wire) to very high level (application semantics), in a single leap. More on the wisdom and execution of that ambition later.

The economic basis for making standards is the concept we call a natural monopoly. This means (at least in this context) that a successful standard will attract and hold all users. Currency is an example: when the state decrees that we will all use a particular currency unit, this becomes a monolpoly. Possessing any other currency means you can't trade, or only at a penalty.

Similar natural monopolies would be rail transport, electricity, phones, the Internet Protocol. You want your toaster to plug into any power socket. You want your phone to reach anyone and be reachable by anyone.

When a successful natural monopoly emerges, thanks to luck or regulation or market forces, it eliminates a lot of waste, also called "friction costs", "transaction costs", or perhaps "excess profits". Natural monopolies can create huge value. Vendors, those selling stuff, have a corresponding incentive to try to capture that value, restoring the profits that would be lost by too much of Adam Smith's invisible hand. The natural monopoly can benefit users, by releasing value. A good example: the Internet. But it can also punish them, by capturing them and then taxing them without mercy. Your mobile phone bill is a case in point.

There is thus often a fundamental conflict of interests when vendors get engaged in standards setting, which is ironic since almost all conventional standards consortia are driven by vendors. How to release all that trapped value and yet catch some of it? In conventional standards consortia, the answer is to use patents and other forms of control. Thus the Digital Standards Organization - disclaimer: I am one of the founders - defines an open standard as one that "is immune, at all stages of its life-cycle, from vendor capture". A neat one-liner that annoyed many people but inspired many others to take a closer look at this conflict of interests.

People sometimes ask, "what makes iMatix different from other vendors, then?" I'd like to explain that we're nicer people but that would be misdirection. Each vendor has the choice of trying to capture the natural monopoly, or competing on quality and performance. The best strategy depends on the size and market position of the vendor. Small, agile vendors like iMatix benefit from competition. Even in a competitive market, customers will pay happily for the best solutions. And, a successful open standard forms the basis for growth into many new future markets.

AMQP addresses a market with huge captive value. The incumbents charge more or less what they like, particularly on the after market for expert services, which is orders of magnitude larger than the upfront software license costs. There are no serious open source alternatives. There are no serious open standards.

So AMQP was born with the mission to go out there and turn messaging into a boring solved problem, a natural monopoly, and release the captive value of the messaging market back to the clients and to future businesses. It's succeeding, in several ways. First, people have bought into this ambitious vision and are taking AMQP seriously as a solution. That is, there seems to be general agreement that AMQP's mission statement makes sense. Second, there are already some very large projects where open source AMQP-based solutions are winning over traditional messaging.

But like all young things, AMQP has some problems. The specification seems to be out of control, getting more and more complex without a clear plan or architecture. AMQP products don't interoperate except in some limited cases. Core protocol aspects are unstable, and more ambitious functionality not happening. The Working Group is getting tired, and many of the original developers have burnt out.

Today, most of my team at iMatix has written off AMQP as a "failure". These are good engineers, like one of my team, who started hacking the Linux kernel when he was 11. He spent two days at the AMQP face-to-face in New York in 2007, and then told me, "Pieter, these are pointless discussions, and if this is how things happen, AMQP will never succeed. If iMatix wants to waste its time and money, fine, but please don't ask me to work on this project, or I will quit."

His immediate and unforgiving critique of the internals of the sausage machine was sad, but not wrong. I was waiting for his opinion to confirm or challenge my own. Shortly afterwards, iMatix decided like some of the other participants to pause all investment in AMQP and wait for things to proceed as they would. Today, the AMQP plan is close to failure, the vision is coming unravelled. Furthermore it's been a predictable slide, clear already in late 2006.

The Working Group discussions are public, now, but discussions of problems are taboo because "it scares potential customers". AMQP/1.0 is promised for end-2008. I'm willing to bet that AMQP/1.0

won't happen in 2008 or 2009 unless it's a rollback to a simpler, older version of the specification.

What is more likely is an slow implosion of the Working Group as it finds itself about to break one more promised deadline, and unable to agree on how to move: forwards, sideways, or backwards.

iMatix has since 2005 invested heavily in AMQP, and in software to run it, like OpenAMQ. We spent so much on this that it almost bankrupted us in past years, not to mention burning out several valuable people on our team. If AMQP does go wrong, we have a serious problem. The risk to my company is so disproportionally high that not doing everything I can to resolve it - including writing this article - would be a failure of management.

AMQP's risk of failure has become increasingly certain over time. As I explained, any potentially lucrative protocol is going to be fair game for the worst kind of fights. As soon as people think, "this protocol means new business", they then think, "how can we kill the competition", without realizing that their competition is helping to actually build their future market.

To some extent, this is a normal part of the process: for AMQP to survive it must redefine and overcome its own fate. As a new open standard in a lucrative business area it was going inevitably to hit the "vendor capture" conflict of interests, and either solve it, or die failing. AMQP's developers and supporters must confront the issues that are preventing AMQP from moving ahead, and solve them. That means actively opening wounds, no matter how painful, examining the causes, and without pity, pride, or shame, make the necessary surgeries.

Making AMQP is not a technical challenge but a social challenge.

My view of design work is, "the more you know about something, the harder it is to solve it." OK, I admit a fondness for cute one-liners. But the point is this: to solve complex technical problems does not demand deep technical competence, in fact that gets in the way. There is lots of deep technical competence around: thousands of highly skilled engineers who know messaging backwards and forwards, and who would enjoy working on something as juicy as AMQP.

The way to solve complex problems is to make it easy for others to solve them. This is where competent engineers often fail: they understand technology but not people. Complexity is a human issue, and good design is about overcoming human limitations, not technical ones.

Paul Gerrard wrote that "software does not obey the laws of physics", comparing software engineering to the building of bridges. My answer, at a conference round-table discussing this quote, was, "software does obey physics, but it's the physics of people, not bricks".

Software standards are no different. The problems with AMQP are all, without exception, about our failure as a Working Group to understand and resolve our own human limitations. Through various combinations of the seven deadly biblical sins we failed to understand the physics of people, and focussed only on the technology of messaging.

The results today are: repeated failure to meet deadlines; ongoing costs with no obvious return; low-quality publications; abandonment of interoperability between implementations; loss of credibility in the eyes of potential users; failure to fill an increasing demand for open messaging;

failure to compete with less ambitious, more flexible grass roots initiatives like XMPP; burnout of many or most key contributors and approaching implosion of the Working Group.

As my mother would say, "Epic fail!"

Some will accuse me of making the problem worse by writing this. Like Brian in his Life Of, my answer is, "make it worse? Make it worse? How can it possibly be worse, I'm already going to be crucified!" The published AMQP specifications are already enough of an indictement, the risk of failure so great, and the price so high, that it's hard to see how anything can be worse than what we already have.

Failure is forgiveable. Not learning and recovering from failure is unforgiveable. Unexplained failure of a highly publicized and rather expensive project is a career killer.

This article will look, hopefully without pity, pride, or shame, at AMQP and what is wrong with it. I hope to help AMQP work as it should have, but also provide insights for reuse elsewhere. In each part I'm going to look at one aspect of the puzzle, and make proposals for fixes.

Keeping It Simple

People who today read the AMQP specifications tend to make one main complaint: the protocol is too complex. This is a justified complaint. Complexity is fatal, it is the garden of thorns around Sleeping Beauty, the minefield and high wall around a technology that keeps people away. A complex AMQP cannot be implemented by small teams, cannot be used in small projects. If AMQP fails in the FOSS world, it will fail everywhere.

AMQP/0.8, the first published specification, had this advantage: it was simple. We saw multiple teams take the spec, read it, and make implementations that *interoperated accurately* with iMatix's OpenAMQ reference implementation, without any prior testing. This is rare, almost unheard of, but shows the power of ignorance (we did not know it had to be hard to interoperate, so we made it work).

Complexity is easy to make, as Leif Svalgaard, a wise programmer, once told me. It is simplicity that is hard. But it's not enough to demand that people make things simple. There is no magic recipe for simplicity, it's not something you can buy or rent or force.

Part of the reason for complexity is that only as one deploys a design in anger, does one learn how to simplify it. We don't know what we need until we've been in the trenches with it, and we don't know how to make it optimally until after that. Many of the original AMQP concepts were over-engineered, solving problems we had theories about but no practical experience with, yet. About 30-40% of AMQP/0.8 was direct waste, but it was impossible to know in advance what those parts were. The 1.0 protocol should have been 90% the size of 0.8, parts chopped out, some new bits added, and with fixes and clarifications.

Instead the AMQP/0-10 specification is many times larger than AMQP/0.8 and little of the real waste has been removed. The 0-9SP1 release will do a lot to address this but that version of the protocol is almost a skunkworks project, a cleanup of the officially deprecated 0-9 branch. The official branch

gets more and more complex, rather than less.

Growing complexity is a symptom of more fundamental problems. Fix these, and you can with luck and hard work get simpler results. So what are the fundamental problems that are causing AMQP to become more complex over time?

The cheap answer would be "sloppy work". This would be both insulting and meaningless. Writing protocol specifications is very hard work and above all it needs clear rules, accurate and focussed vision, and strong will. This is notoriously difficult to get in a committee. However, I think AMQP has been derailed, so far, just by bad instructions from some of the leading voices on the Working Group.

Like King Midas, who asked that everything he touched would turn to gold, it's easy to make broad wishes that turn out to be disasterous if they are taken literally.

In my view, the first problem is that AMQP has been positioned as an "Enterprise Technology". What does this mean? I think the intention is that technology is like air travel, which is segmented into a cheap, sterile, tiresome, squashed experience for the ordinary jeans-and-sneakers proles, and an expensive, luxurious VIP experience for the high-value suits. You have ordinary prole technology, and then you have Enterprise Technology. AMQP has to be the very best, since it's going to be used in banks and so on. You would not want your pension to be lost thanks to a missing transaction. AMQP users are VIPs, and intend to travel Business Class. Both customers and suppliers prefer the luxury route: it means bigger budgets, and more air miles.

Now this may make sense in air travel, but as far as I can see in the software world, the prole technology is generally better, faster, more secure, more robust, and above all, simpler. Proles have less money to spend, but most importantly, less money to waste.

In the software world, the modern proles are of course FOSS developers and users. It's a world I know well - I've been writing free software code generators and tools since 1992 or so. We don't see our software as less good than closed-source products - but the opposite: disclosure makes it impossible to hide mistakes. Cheaper is better, the proles travel first class with unlimited champagne and neck rubs, at Mach 10, while the Enterprise clients are charged a hundred times more for the privilege of standing all the way to Casablanca in a crowded toilet with seven screaming kids.

AMQP's positioning as Enterprise Technology has made the Working Group tolerant of complexity that would not have survived one hour on the Internet. People would have said, "OMG? ROTFL!" and posted the proposals onto Slashdot where they would have been mocked six feet under. Instead they got incorporated into the official specifications and released as Gospel. As a form of subtle comedy ("we're not ready yet, so here is some light farce to entertain you"), it would have a certain style. As serious work and part of the historical record, it is a failure.

Apart from the tolerance for shoddy work wrapped up and oversold as "extra value", Enterprise Technology sets the bar so high that no-one can jump it. "We must have 100% guaranteed reliability even if the server crashes", sounds fine but this single demand - repeatedly stressed and sold by some of the AMQP participants - has caused most of the AMQP slippage. In fact no vendor can guarantee 100% absolute reliability, not in messaging, not in any software, not in cars, food, computers,

anywhere. And no-one needs it. As long as the probability of loss is low enough, that's fine. Or, put it like this: if it's cheaper to compensate a client for a lost message/exploding computer/late flight/failed brakes than to make the technology more reliable, stop making it more reliable. The better is the enemy of the good, and it's more costly too.

Yet as far as AMQP is concerned, Enterprise Technology must be 100% reliable. This means making transactions that can properly survive a failover from a primary to backup server. This means the protocol must have mechanisms for switching sessions over from one server to another. This means we need to redefine the atomicity of every operation from queue creation to message transfer, as well as redefine what a session is. That means we need to treat everything as a message. Or perhaps the opposite. That means we need dozens of new data types, including multiple variants of that insufficiently complex concept called a "bit". Yes, finally we get to a 300-page specification. Now we have something suitable for the Enterprise!

Well, no. What we have is a smelly mess looking like Godzilla's regurgitated breakfast. You can recognize pieces of Tokyo, but it's not a place you'd want to live in anymore. In communications, reliability comes mainly from making servers that don't crash. And this comes from making pieces that are simpler. And this comes from lowering the bar on the demands for perfection. Ironically, the ambitions of AMQP to be perfect have the opposite effect.

I'll explain my own views on reliability later. They are not comfortable reading for the current AMQP world view, since they refute the assumption that AMQP servers need to be big, complex, and reliable, but I've not heard evidence-based counter-arguments. However, whatever the specific solution, the AMQP Working Group must learn to accept "good enough" when it's there.

Now, the second big reason for AMQP's march towards perfect entropy: "there can only be one version, ever!"

This has been the mantra of some of the larger participants. From the customer perspective, the cost of upgrading a wire level protocol is huge. HTTP stopped at 1.1 and never improved. Therefore, AMQP/1.0 must be perfect, because it will never be changed again. We get one chance, we must get it right!

From the vendor perspective, an increasingly complex protocol dominated by a single team creates a lucrative opportunity for capture. Deliberately or accidentally, AMQP/0-10 has become a dump of the internal structures of a single implementation (Apache Qpid). This is great: when AMQP/1.0 finally hits the market, there will be just one (surviving) implementation that can capture the entire market! Too bad for the competition, but this is business, not charity.

So there has been a constant coalition of interests between the larger users and larger vendors that is forcing AMQP to march on towards complexity and chaos, and is suppressing dissenting voices that have called since 2006 for simplicity and closure. But the coalition of interests, though loosely logical, is based on flawed reasoning.

Let's start with the theory that we only have one chance to make AMQP. In my 30 years of making software, I've noticed something: change happens. AMQP was originally written to survive 50 years (which sounds long but not if you consider it make take 5-10 years to actually make the protocol). If

there is a 1.0, there will be a 2.x, 3.x, 4.x, and so on. The reason is simple: a basic networking protocol has to evolve to keep up to date with new technological opportunities. This is why my notebook supports four or five generations of Wi-Fi.

It's no big deal: the protocol has a version header and peers negotiate the version they want to use. Servers and client libraries can handle multiple versions if they want to. It's not pretty but it's not a new or particularly difficult problem.

It starts to look obvious that AMQP should have been released almost as-is in 2006 as version 1.0 and increasingly difficult to justify not releasing the currently most interoperable spec (the sadly named but clean and "good enough" 0-9SP1) as "1.0".

If I was giving advice to the AMQP Working Group, I'd recommend releasing 0-9SP1 as 1.0, and insisting on interoperability from all vendors. Then, we can fix other issues and move on to 2.0. We know 1.0 won't implement the Enterprise Reliability some of us dream about. Fine. If it's doable, we can do it later. If it's not, we should not kill the baby just because it does not fly like an angel.

This proposal will cause serious pain from the Qpid team, who have moved far away from the 0-9 tree. But it seems clear they will need to bite the bullet and take the pain. They have no choice: they are alone, with an almost proprietary version of AMQP that looks more like a dump of their software format than a properly designed specification, and no interoperability with any other vendor. They will have exactly 100% of a market that will be worth close to zero. It is a bad business bet, guaranteed to lose large amounts of money. The real profits are where the competitors are, for that is where customers will put their dollars.

It's like the old tale of where to build a profitable Chinese restaurant. The answer: build it where all the other Chinese restaurants are, because that's where the clients go. Thus you get a nice, lucrative Chinatown. Alternatively, you can build a Chinese restaurant all alone by itself, and attract clients who actually hate Chinese food but were hungry and in the area. As they demand steak and fries and beer, and eat the delicately carved carrot dragons, you can reflect on the mainly empty dining room and idle staff, and wonder when your funds will run out.

The AMQP Working Group won't make the decision to commit to the 0-9SP1 specification easily, because it means great loss of face. There have been Big Promises of What Will Be Delivered in 1.0. Grand promises, over-sold with elegant presentations to important people. A lot of credibility is at stake and I've rarely seen corporate pride bow to reality except in times of profound crisis. So today it seems more likely we will end up in the scenario I drew in the first section: failure to deliver a convincing 1.0 spec, collapse of confidence, argument, blame, and inability to decide over which way to move.

Still, just because a mess looks likely, does not make it inevitable.

In this section I've looked mainly at the need for palative measures: how to reduce the pain of what is in any case going to feel like a death in the family. Step 1: eat a big dose of humble pie and accept that we can't, yet, make a perfect package. Step 2: release the best we made so far as 1.0 and demand support from vendors. Step 3: think carefully about how to proceed.

I'll start to look at step three in the next section, by examining some of the most traumatic experiences our team went through over the last three years as we tried to contribute to AMQP, and gradually found ourselves edged out and burnt out.

Pain is not, generally, a Good Sign

In the previous sections, I explained the reason for this article, and made some obvious if annoying recommendations for salvaging some of the work done so far. Putting pride aside, AMQP can be moved to a Safe Place simply by releasing the 0-9SP1 text as 1.0 and putting pressure on all implementations to support this. To err is human, but to admit it and be prepared to backtrack requires divinity, or a really good parser.

However, securing the work done so far - which is only partly on AMQP itself, and more significantly on deployment of AMQP - is only half the battle. Yes, it would be great to at least get closure on this. But what happens next? I said AMQP needs to aim at a 2.x, 3.x, 4.x release over 50 years. How?

I'm not a messaging expert but I do have the gift (curse, maybe) of being undiplomatic and more often than not accurate about things that don't work. Having worked in hundreds of teams, of all kinds from professional to voluntary, has taught me a lot about the best and worst ways to organize. The AMQP Working Group is better than some, but worse than others.

My only real yardstick is my own pain level, and the observed pain levels of those I work with, like my colleague at the New York face-to-face. If an organizational model hurts me, or hurts my fellow participants, then there is something wrong with it. Pain is not, generally, a good sign.

One essential difference between different participants in the AMQP Working Group is how they perceive pain. Some are large, wealthy, and used to the complex and difficult processes which drive larger organizations. They presumably feel pain all day, and appear to be pretty tolerant of it. But other participants are small, modestly funded, and tend to work with light, agile organizational models. These are the canaries in the coal mine. It's the same reason I want AMQP to be attractive to FOSS developers. The smaller the player, the more sensitive they are to problems that ultimately hurt everyone.

Presumably everyone in the AMQP Working Group has their own views on this, but for iMatix there are three real causes of pain that have forced us to take distance and at some level, treat AMQP as a problem and risk rather than an opportunity.

The first cause of pain is that AMQP is one document (actually, one written text and one XML file). The motive was that interoperability would be easier with a single indivisible specification than with a stack of specifications. But that single spec is now about 300 pages, and growing. Some specifications are huge, but if I compare AMQP to the IEFT RFC documents, it's clearly over-sized.

Before we published AMQP/0.8, the XML specifications consisted of a top-level file, and then one XML file per class. This made it simple to edit individual classes, and since the protocol could be expanded and modified by adding or removing classes, it made it easy to manage. But this was

considered too difficult for XML parsers, so this structure was flattened into one file. I had argued that a tiny Perl script could do this flatenning on demand for code generators, but no: the protocol had to be One File.

A large single document, XML or text, is not just a nightmare for editing, it does not make sense for a specification that addresses, like AMQP, many problems at many different levels. For example, compatibility with the Java Messaging System (JMS) is another of those Enterprise Technology check box items that interests no FOSS developers but seems to possess some of the AMQP participants like an evil spirit possessing a weak-spirited priest in a Nollywood horror flick. Why is JMS compatability, a legacy high-level application language-specific feature, defined in the same text as wire-level encoding of a 4-byte integer, a low-level feature? Or, to take another example: why is the AMQP specification for connection negotiation in the same document as the specification for distributed transactions?

Comingling concepts like JMS, basic data types, connections, and transactions in one document makes it possible for people to propose - and this has happened often - changes that cut across many levels at once, as if there was no architecture at all. "Hey, let's make connections part of the transaction and give them JMS headers that are Java compatible!" Sounds great, no? And pretty soon, basic notions of architecture and structure are gone, and what remains cannot be improved, only reformatted with a nicer font and page layout.

I see no defensible reason for aiming to make AMQP a single document. AMQP's functional scope hugely exceeds the bounds of a single specification and already we have multiple other specifications (for remote administration, for federation, for message bundling, etc.) that don't sit inside that single specification, so can't even exist, on paper.

We need to redesign AMQP, after 1.0 is released, as a stack of interoperable protocols, each defining one layer of the stack, and allowing competition and experimentation at each layer. Which brings me to the second major cause of pain: ownership. Or rather, the lack of it.

My own distaste with the AMQP process came when I saw good, simple concepts being reinvented, broken, removed, and replaced with what seemed immature complexity, for ill-defined reasons and with little or no space for dissenting opinion.

The process seemed to go thus: "We tried implementing the existing spec but frankly we did not like it, so we wrote some new code, and here is a dump of our design. We will now proceed to force through this no matter how much argument it takes us. Resistance is futile." Shamefully, the AMQP Working Group stood aside and allowed this to happen, in case after case, as if the spec had no weight, and represented no investment, except as a disposable straw man.

The AMQP/0-9 specification is a rich source of truly painful examples, helpfully marked as "Work in progress" for the reader. Ask yourself what process allowed such a document to be published, and "lack of ownership" is a large part of the answer.

I will take one early example. Why do we write 0.8 with a dot and 0-10 with a stripe? Originally the AMQP version header had one byte for the major version and one for the minor version. The minor version was encoded as a decimal number, two significant digits from 00 to 99, so 0.8 was:

```
major = 0
minor = 80
```

Which allowed us to make updates to minor releases, like 0.81. It would have made it impossible to make a 0.10 release (since that would be the same as 0.1). A good thing, you might be thinking. Ninety-nine minor versions is surely enough.

After one of the first big flamewars in the Working Group, and after what seemed like a thousand emails between two sides, the version number was changed to encode the minor version as a cardinal number (1 to 255). This made it possible to imagine 255 releases before 1.0, but made it impossible to specify an update to a minor version.

There was never a clear reason for this change except, "we don't like the current scheme, so here is a new one". No clear explanation of what was broken, just argument over opinion until the louder voice won. At the time I complained to the Working Group leadership that we were allowing change for its own sake, but I had little support. People who could have kept order were either unwilling to enter into the discussion, or felt it was necessary to allow it to happen. But what actually happened was a failure, and the first of many.

So, today, we write 0-9SP1 rather than 0.91, and we have to negotiate the differences between 0-9 and 0-9SP1 *after* the official protocol negotiation, by putting magic values in one of the connection methods. This means that when we actually need simple and robust protocol negotiation we can depend on, we don't have it.

There is not even modest pleasure in being right, and my explanation is not a "told you so". My point is that the process that drives AMQP is fundamentally unhealthy, and that sub-par results are not exceptional but dominant. I hope that my explanation of how this process has come to exist, and how it fails, is objective even though I was involved in many of the arguments.

The "loudest wins" process would not survive if there was a clear notion of ownership.

No-one owns any part of the protocol because AMQP is not divided into ownable pieces, and because there are no rules to protect such ownership.

Let me clarify what I mean by ownership of a specification. Obviously it's not about legal ownership. It is about blame and responsibility. If a piece has someone's name on it, that person has rather strong incentives to make it work. If a piece is owned by a committee, the incentives to improve it and protect it are just not the same.

AMQP is one over-sized package owned by a committee.

Assuming the architecture of AMQP was a stack of protocols (like any comparably complex IETF specification), then we could see some nice things happen:

1. Each layer of the stack could be redesigned by anyone. Think of SASL security mechanisms. You want to make a new one? It's clear where, and how, to do the work.
2. Each piece of work could be properly tied to its authors, persistently. Think of the IETF RFC model, where each RFC has an author's name at the top. Credit, you might think. But blame and

responsibility seem more pragmatic mechanisms for quality.
3. There could be a proper process, an etiquette, for changing any given piece. That's not feasible with huge pieces of work.

The lack of a polite change process seems to have affected many participants, so not only does the protocol get progressively worse, we lose those people best able to fix things. It is an incompetence-complexity tarpit, and AMQP is sinking into it like a lost beast.

A healthy change process, in software and in standards alike, is tightly linked to ownership and works as follows:

1. You need a change in a layer you depend on.
2. You identify the breakage or lack.
3. You identify the piece that is broken.
4. You identify the author or authors of the piece.
5. You notify them of the problem and ask for a solution.
6. If you are able to, you propose something yourself.
7. You wait for a reasonable time and hope your question is answered.
8. If necessary, you bribe, blackmail, cajole the authors.

And, if the authors don't respond, insult you, mistreat your contribution, or otherwise behave anti-socially, you take their work, fork it, start a flame war, and try to win people over to your - you hope - improved version. At that point you become responsible for that piece. You saved it, you have to feed it.

This is how successful open collaborative of any kind works:

- A good modular architecture that splits a large problem into smaller pieces that collaborate and compete in clear ways defined by formal contracts;
- A clear ownership model in which every piece is ownable and owned;
- An etiquette for changes that relies on and respects ownership, while making it possible for forks to be made freely.

The goal is to take the natural tendencies to compete and collaborate, and turn these into useful drivers rather than reasons for conflict. The same rivalry that can spawn a thousand flames can drive competing teams to superb achievements. But only if the social architecture is done right.

I've said that AMQP is a social, not a technical, challenge. Social architecture goes much wider than organization of the Working Group. It extends into AMQP users, and contributors outside the core specification developers.

In the next section I'll look at another strong indicator that something is Not Right with AMQP: the lack of an online community around the technology.

Where's the Meat?

So far in this article, I've shone the light on some of the less fun parts of how AMQP is being made. I mentioned that there are IMO thousands of people who would love to, and are highly competent to, participate in a project like AMQP.

After two years of public AMQP releases, where are these contributors? Where is the community? Where are the forums, the blogs, the FOSS projects, the normal symptoms of a healthy standards process?

Their almost total absence is perhaps the most solid sign that AMQP is not in a good state. The market is rarely wrong, and while many people appear to believe in AMQP's potential, few or none of those thousands of potential protocol engineers have signed up and put their time and money into the specification game. The spec is being developed by software engineers working for large firms, not protocol specialists.

On-line communities, it seems obvious, grow. What is less obvious is that they seem to grow as a response to social challenges. For example, I'd argue that the Wikipedia community is strong and confident *because* it is continuously confronted by trolls, spies, liars, and manipulators, and it's always developed ways of beating them. The fact that anyone can edit a Wikipedia page and thus make a mess of someone's neat work is not a problem. Rather, it is the fundamental driver of Wikipedia's success. Edit wars create emotions that bind the community together.

So from one point of view, the AMQP community needs problems to solve, preferrably social problems, not technical ones. In other words, exactly the kinds of problems that have beset AMQP's progress.

The lack of a community is both symptom, and in my view, contributing factor. There is almost no AMQP community, only a self-selected and weakly diverse expert group.

If any excuse was needed for exposing AMQP's inner turmoil and thus embarassing people, it is this: documenting and exposing problems makes it possible for others to solve them. And of course, social issues, the physics of people, are never tidy or painless.

However, the big difference between AMQP and Wikipedia is that the latter is a meritocracy where anyone can join in, individually, and be promoted on the basis of their proven works. AMQP is a club of businesses, one has to be invited, and one represents one's company, not oneself. The rules for joining and for voting are being relaxed somewhat but it's still a tortuous process.

At a certain stage the number of AMQP participants was restricted simply because the effort of sending around packs of contracts was too great.

The main reason for this heavy layer of contracts was fear that one of the participants might introduce patented technology and thus try to capture some of that lovely natural monopoly efficiency. It's a reasonable fear, but there are simpler ways to resolve it.

First of all, multilateral agreements - between many parties - are not scalable. AMQP needs to be hosted by a proper not-for-profit organization so that agreements can be one to many, and the cost of welcoming new participants is linear, not exponential.

Secondly, the contracts can be a lot simpler. Basically they are a promise from a contributor - individual or organizational - to whatever entity owns AMQP to not patent anything in the specification, and not introduce anything that is patented, except under condition of a royalty-free, global, irrevocable license to anyone wishing to use it. There is some stuff about copyrights and trademarks, and that's it.

Thirdly, the AMQP development contracts contain a large amount of detail on how the Working Group should operate. Now, agreeing on policies for making changes and voting on them is obviously a good thing. Putting these into the one place it's almost impossible to change is not a good thing. Policies need to be lightweight and easy to improve. As it is, the heavy contracts we all signed about the mechanisms for collaboration have not helped a jot when it actually came to resolving conflicts.

Apart from the procedurable barriers to entry, AMQP's technical complexity makes it very hard for anyone not already in the loop to take part. I will repeat an old joke - if you know what any part of the latest AMQP specification means, you were probably the author.

Rapidly increasing complexity is a sign that the process is failing.

But even when AMQP was still simple, there was no effort by the Working Group to build an independent community. It felt like fear of scrutiny: discussions were kept private, the public Google groups some of us had set-up at the start were not used; the fast and simple wikis some of us tried to get going were abandoned in favour of horridly slow Enterprise (yes) wikis poorly hosted by one of the participants, who insisted on SSL protection (for "Enterprise security measures") for every page but was unable to install a valid SSL certificate.

At iMatix we're used to setting up community infrastructure. One of my projects, which is hosting this article, is Wikidot.com, a free wiki farm designed exactly as a fast and flexible community platform. Community infrastructure is my business. For AMQP we tried, repeatedly, to set up a simple and pleasant infrastructure, and each time we were stopped by voices on the Working Group who felt there were better ways - which never emerged.

When it takes thirty to sixty seconds to make a single change to a wiki page, you can be sure that people stop contributing really quickly. A cynic might add that the slow SSL interface and multiple login pages only bothered people outside that particular company, not engineers from inside it, so that it was not surprising that drafts of AMQP increasingly came to reflect the views of a single player.

One of the ways to capture an emerging standard is to capture the processes, and infrastructure behind it. AMQP is not safe from this, but it must become so.

Community building needs:

- An entity that is independent of vendors and their antagonism towards outside expertise.
- Suitable funding to pay for community development, system administration, etc.
- Conscious marketing towards pioneer experts who like new technologies.
- Events, newsletters, forums, blogs, and so on, aimed at the community.

And above all, a simple way for independent experts to participate in the process, and contribute with

some confidence. Which brings me back to the "Pain is a Bad Sign" principles of a good modular architecture, ownership rules, and a clear and ethical change process, all of which are lacking in AMQP and will exclude any community participation except at the margins.

If I was giving advice to the AMQP Working Group I'd suggest them to focus less on writing specifications, and more on building the successful community that can write the specifications.

The cardinal sin of any expert is to believe in their expertise. We succeed only when we recognize and expect our limitations and compensate for them. It's our mistakes and failures that should teach us the most. And speaking of failures, the basic AMQP design, which I am responsible for, has very large bug in it, a massive design flaw based on wrong assumptions and driven by premature optimisation. No-one seems to have spotted this design flaw, perhaps because all the engineers still working on AMQP share the same wrong assumptions. An insufficiently diverse group sometimes can't spot the obvious.

In the next section I'll explain what this flaw is, and how fixing it will open the door to a faster, simple, and altogether more enjoyable AMQP experience.

Premature optimisation is the fast lane to hell

While I'm not going to go into technical detail on AMQP (partly because I've not tried to follow the tidal wave of changes in the 0-10 family), some design decisions are fundamental, and if they are wrong, they affect everything.

If you've looked a little at AMQP, you'll see that it's a binary protocol. This means that methods like Queue.Create are defined as binary frames in which every field sits in a well-defined place, encoded carefully to make the best use of space. Binary encoding of numbers is much more compact than string encoding. "123" needs one octet in binary and three as a string. Because AMQP is a binary protocol, it is very fast to parse. Strings are safe, since there is no text parsing to do. AMQP's binary encoding means AMQP software is not at risk of buffer overflow attacks. And it's easy to generate the encoding/decoding routines, since AMQP is defined as easy-to-parse XML.

Overall, AMQP's binary encoding is a big win.

That last statement has been proven to be wrong. Evidence shows that AMQP's binary encoding is a fundamental mistake. And it's a mistake I take full responsibility for: this was my design, and I was very proud of it, for it was nicely done. An expert mistake. To understand why I'm admitting this, let's look at the advantages, and costs, of this approach, and let's deconstruct those basis assumptions that I claim are now proven to be wrong. Finally, let's compare this with an alternative approach based on more accurate assumptions.

Advantages of binary encoding:

- It is more compact.
- It is faster to parse than a text format.
- It is safer to parse strings.

- The codecs can be fully generated.
- It is easy to process in silicon.

Costs of binary encoding:

- You need codecs in the first place.
- It creates endless incompatible versions of AMQP.
- It is more complex to understand and use than string encoding.
- There is a lot of emphasis on data types.
- Even the simplest client API is significantly complex.

Now, a fast, compact wire-level encoding is surely worth the hassle. After all - so went the argument - AMQP is designed for speed, and it's Enterprise Technology, and complex APIs are not a big problem. It's true that text-based protocols like HTTP can be played with using a TELNET client but in reality no-one writes HTTP clients, they use existing libraries.

So conventional AMQP wisdom is that the costs of binary encoding are a necessary price to pay if one wants a fast, reliable protocol. Our view was that we could not hope to achieve the necessary performance over a text-encoded protocol like HTTP.

The main assumption underlying AMQP's encoding is that it's necessary for performance reasons (speed of parsing, compactness of data). If I can show that this assumption is wrong, I have demolished the main justification for binary encoding.

Here is a pop quiz to test your knowledge of protocols. What is the fastest common messaging protocol, built-in on every modern operating system, integrated into every browser, and capable of saturating ordinary networks? It is faster than HTTP, and much, much faster than AMQP. In fact if implementations of this protocol were not dependent on reading and writing everything to disk, they would probably score as the fastest messaging application every designed.

The answer is FTP, the humble file transfer protocol, beloved of network engineers who want to check whether a network link is configured for 100Mbps or 1Gbps: FTP is capable of shoving data fast enough down the line to prove without doubt how fast the network is.

Now, the interesting part and the reason for my question. What is special about FTP that lets it transfer data so rapidly? And what lessons does this provide for AMQP?

Incidentally, the origin of my views on AMQP performance come from our work on ZeroMQ, a messaging fabric that can transmit millions of messages per second. ZeroMQ is so fast because it uses the same techniques as FTP, rather than those used by AMQP.

FTP wins because it uses one connection for control commands, and one for message transfer. This is something that later protocols, like HTTP, did not do. But FTP is mostly faster and simpler than HTTP. Faster and simpler are desirable features.

AMQP's main assumption that binary encoding is needed can be broken into more detailed assumptions, each wrong:

1. *That it is neccessary to optimise control commands like Queue.Create*. The assumption is that such commands are relevant to performance. In fact, they form a tiny fraction of the messaging activity, the almost total mass being message transfer, not control.
2. *That control commands need to occupy the same network connection as messages*. The assumption is that a logical package of control commands and message data must travel on the same physical path. In fact they can travel on very different paths, even across different parts of the network.
3. *That the encoding for control commands and for message transfer need to be the same*. In fact, there is no reason for trying to use a single encoding model, and a big win from allowing each part of the protocol use the best encoding, whatever that is.

What AMQP should have done, from the start, was to use the *simplest possible* encoding form for commands, and the *simplest possible* encoding form for messages. I can't over-emphasise the importance of simplicity, especially in young protocols that have to support a lot of growth.

The simplest possible encoding for commands is in the form of text, with (for example) the 'Header: value' syntax that is well known from HTTP, SMTP, etc. This is trivial to parse using regular expressions. Attacks on this kind of encoding are in the form of oversized strings, and they are easy to deal with. There are no funny data types, everything is a string.

Using a simple text encoding for commands releases AMQP from many of its shackles:

- It becomes obvious to developers.
- It becomes easy to have backwards compatibility.
- It becomes easier to write clients.
- It becomes easier to debug and test AMQP test cases.

Does the text parsing create a performance penalty? Yes, but it is absolutely irrelevant - and I can guarantee this - in the overall performance question.

What then is the simplest possible encoding for messages? AMQP defines a rather impressive envelope around messages (around 60-100 octets), which may be fine for large messages and low performance goals, but is bad news for small messages and high throughputs. When we developed ZeroMQ, we wondered just how small the message envelope could get. The answer is quite surprising: you can reduce it to a single octet.

The simplest message encoding has a 1-octet header that encodes a 7-bit size and a 1-bit continuation indicator:

```
[octet][0 to 127 bytes of binary data]
```

Empirical tests also show that this is the most efficient encoding for random message sizes. We can of course define other encodings, each with their own cost-benefit equations.

Now, a necessary question is, "how do we mix those simple text-based control commands with that simple message encoding?" There are several answers:

1. We can wrap binary messages in textual envelopes. This is how BEEP and HTTP work. This

single-connection design looks simpler but in fact becomes quite complex, and it is inefficient.
2. We can use distinct connections for control commands and for messages, like FTP. This is simple but means we need to manage multiple ports. This feels wrong: it should be possible to do everything on the single AMQP port 5672.
3. We can start with a simple text-based control model and switch to simple binary message encoding if we decide to start message transfer. This is analogous to how TLS switches from an insecure to an encrypted connection.

I prefer the last option. In any case, it is useful to separate control and data. Mixing them, as AMQP does today, creates some extraordinarily delicate problems, such as how to handle errors that can hit both synchronous and asynchronous dialogues. AMQP's exception handling is an elegant solution but wouldn't it be nicer to have something more conventional?

There is a concept I call "natural semantics". These are simple patterns that Just Work. Natural semantics are like mathematical truths, they exist objectively, and independently of particular technologies or viewpoints. They are precious things. The AMQ exchange-binding-queue model is a natural semantic. A good designer should always search for these natural semantics, and then enshrine them in such ways as to make them inviolable and inevitable and trivial to use.

The natural semantic for control commands is *pessimistic synchronous dialogue* in which every request is acknowledge with a success / failure response. The natural semantic for data transfer is *optimistic asynchronous monologue* in which one party shoves data as fast as possible to another, not waiting for any response whatsoever. I'll answer the question of "what happens if data gets lost" in the next section.

AMQP does allow both synchronous and asynchronous dialogues but it's not tied to the natural semantics of control and data. The natural semantics are weakly bounded, insufficiently inevitable. And these weak boundaries are fully exploited as people experiment with asynchronous control and synchronous data, creating unnatural semantics.

HTTP is slow because it uses the wrong semantics for data transfer. Wrapping data in control commands, as BEEP does, would make the same mistake. Using two separate connections is good, because it cleanly separates the two natural semantics. But if you've ever implemented FTP servers and clients (I have, and they are evil in this respect) then you'll know that FTP's port negotiation, which is designed to cross firewalls, is a big part of the problem. We don't need this in AMQP, we can use the single AMQP port for all connections and indicate at the very beginning: "this is control", or "this is data".

While we're at it, let's forget the whole notion of connection multiplexing, called "channels" in AMQP. This solves HTTP's problem, where clients open and close many connections in parallel as they fetch the components of a web page. AMQP clients open one connection and keep it open for ages. Multiplexing solves a non-issue and does it quite expensively.

Let me wrap this up in a single statement: my basic AMQP design incorporated a set of seriously flawed assumptions that made AMQP *significantly* more complex, difficult to understand, incompatible with itself, and slower, than it should be. I am so sorry! In my defense I'll point out that no-one else has pointed out the flaws in these assumptions, so they cannot be that obvious.

In the next section I'm going to point at an even larger assumption, one that underlies the whole AMQP vision, and one that I've always felt uncomfortable with. I'll argue that it too is flawed.

On avoiding special cases

I've looked at what I believe are the reasons why AMQP is too complex, why it has been painful for most of those involved, why there is no community around the protocol, and how and why as the original AMQP author I almost totally misdesigned the wire level framing.

These may seem like serious charges and admissions, but they are all both natural and recoverable. In this section I'll make more a fundamental critique of the AMQP vision. I have to admit that my views on this particular topic pit me directly against others in the AMQP Working Group, who must think I am either naive, trollish, or just wrong-headed. Yet I've been forced into my particular point of view, which was not where I started with AMQP, by the weight of evidence.

Mainly, I'll argue that the vision of AMQP in which a central server reliably stores and forwards messages is wrong, based on two mistaken ideas. One, that we *de-facto* have a central server. Two, that we have a single reliability model. I'll try to explain where these ideas came from, and why I think they are wrong.

AMQP has many sources of inspiration but most of all, it was inspired and shaped by the notion of a central server providing functionality roughly equivalent to JMS, the Java Messaging System. "Inspired by" does not have to mean "is a literal copy of". The Kevlar vest was inspired by the sub-machine gun. The AMQ model of exchange-binding-queue is the Kevlar to JMS's "destination", which an example of a perhaps perfectly unnatural semantic sold as "Enterprise technology".

But AMQP does, more or less, aim to cover the same functional ground as JMS, and if you look at AMQP's definitions for transactions, acknowledgements, message delivery, and message headers, you will see many echoes of the JMS specification. We were trying, in part to make it easy to support JMS later, and in part, just reusing concepts that we assumed worked, or at least worked well enough to take us through to the next version of the protocol.

Let me recap some of the relevant assumptions we inherited from the JMS specifications and the JMS products we felt we were competing with:

1. That there is a central server or fail-safe central cluster of servers. This is conventional wisdom, especially in the Enterprise, which seems to like big central boxes.
2. That the protocol must support "fire-and-forget" reliable messaging. This is a logical assumption, since if reliability is not in the protocol, every implementation will make its own version, and we won't get interoperable reliability.
3. That there is a single, ideal model for reliability, which looks a lot like relational database transactions, and that this single model can handle all application scenarios.
4. That such reliability must be implemented in the central server(s). This is logical since full reliability needs things like horribly expensive Enterprise-level storage area networks (SANs), which obviously need to sit in the middle somewhere.

5. That such reliability is implemented by conventional transactions, operating on published messages and on acknowledgments. This is a direct lift from JMS, which represents the way successful products like MQSeries do it, and so must be right.
6. That these transactions must survive a crash of the primary server, and recovery on a backup server. This is just a consequence of the previous assumptions. But it's the stinger.

Making JMS-style transactions is not very hard, we did this in an earlier version of OpenAMQ (around AMQP/0.4), but ripped it out, because it was so ugly and we did not need it for our target application.

What is hard is making (centralized) transactions that can survive a server crash. Many of the changes in AMQP/0-10 mystify me, but as far as I can understand, the determination to get transactions that recover reliably from crashes is driving a lot of it (and JMS compatibility most of the rest).

In other words a large part, perhaps most of, the work done on AMQP over the last two years has been focussed on getting this "Enterprise level" reliability. My best understanding is that this work has been driven by large corporate almost-clients who absolutely insist that they cannot commit to AMQP (excuse the pun) until it delivers this very desirable functionality. It feels a lot like belief-based investment, rather than evidence-based investment.

Transactions - the conventional unit of reliability - also fit very uncomfortably on top of asynchronous message transportation, which is the core of AMQP. Just when is a message delivered? Is it when the exchange has routed it, when it's been put onto a queue, or received by an application, or when it's been processed and acknowledged? What if the message is routed across multiple servers in a federation? What if we want to use a multicast protocol? If reliability is going to be built into the basic protocol then it must have answers to these questions. I have not seen answers.

If we look at the latest drafts of AMQP, they seem to be telling us clearly, "this problem is too hard to solve". Some of the AMQP editors seem still to be optimistic but I don't see the basis for that optimism, and as far as I can see, AMQP is not going to deliver "fire and forget" reliability until there is a radical change of strategy. I'm not against the notion of fire-and-forget. But putting this into the wire-level protocol has not worked, despite effort that is now several orders of magnitude greater than the work of originally designing AMQP.

When I wrote about reliability in the section, "Keeping it simple", I was referring to this. A high-level feature demand that creates disproportionate complexity at all levels of the protocol must be questioned, especially when it turns out to be approximately impossible to implement.

My view is that trying to make perfect reliability in AMQP, today, is mixing innovation and standardization, which is like mixing petrol and fire crackers. AMQP is not, though it should be, partitioned in such a way that reliability can be layered later on top of a more basic protocol that is locked down today. Big changes make a mess of everything, and that mess cannot be cleaned up, since it is structural.

There may be several reasons why it has been impossible to make reliability work in AMQP. Perhaps the very notion is flawed. Perhaps we've taken the wrong approach. Perhaps the lack of architecture in AMQP makes it impossible to do such work, because every change breaks numerous other things. All

these may apply to some degree, but in my view, the reliability issue has been impossible to solve purely on a technical level because we've been trying to solve a restricted special case.

Perfect interoperable reliability may be solvable, but only if we deconstruct our most basic assumptions about the role of the AMQP server, about centralization, and about what "the protocol" is.

We'll start by looking at performance, and the impact of a central server on performance. Forcing all messages to pass through a single point creates a performance bottleneck that gets worse as the number of clients increases. It adds two kinds of extra latency: first, the cost of reading, processing, and writing the message. Second, the cost of waiting as messages queue up to be processed. As volume through a central server increases, latencies get exponentially worse and worse, which is a nightmare for serious messaging users, who need reliably low latencies.

Protocols like IP have similar problems and they solve them brutally: when too much data arrives on the network, throw it away. But since AMQP is aiming to make a fully reliable protocol, this is not an option, so we are left with an unsolvable problem.

IP's solution has several desirable consequences. It lets any point on the network take charge when it is overloaded. It lets network traffic flow around failures. It lets networks scale to anything from zero to hundreds of intermediate routing points. If IP could be described as a "optimistic/cynical" protocol then AMQP is today a "pessimistic/idealistic" protocol. IP hopes data will get through but does not assume it will. AMQP thinks data will get lost but believes it can prevent that from happening.

A central server that routes messages also halves the network capacity, since every message must be sent twice: first from the publisher to the server and then from the server to the client.

So pumping messages through a single central point is bad for performance and scalability. But from a more general network design perspective, it is a rather special case: every message goes through exactly one switch point.

If we believe in centralization, that means we want the number of switch points to be exactly 1. Let's call this the "N = 1" case. But belief must take a back-seat to evidence. Do we have cases where N is not 1, and are these cases relevant?

Certainly, we have real-life cases where N is 2 or 3, so-called "federation" where one server acts as a client of a second, which may itself be the client of a third. Federation is an essential architecture: it solves the problem of how to extend AMQP networks beyond a single local network. We have used federation in real-life cases and we can say empirically that "N > 1" is relevant.

How about N = 0? We can look at ZeroMQ, which pushes routing to the publisher edge, and queueing to the consumer edge. With no central server, and with a lot of careful coding, ZeroMQ can hit speeds that are an order of magnitude greater than any AMQP implementation. And as you'd expect from a peer-to-peer model, it continues to scale smoothly as the number of peers on the network grows.

I don't believe performance is optional: it is so important that any design which pushes it up by a significant factor must be taken into account. After all, we spent a great deal of effort optimising

(wrongly, it turned out) other parts of the protocol. Consequently, "N = 0" is also relevant.

It seems clear therefore that we must address N = {0, 1, 2, 3,…}, with an architecture that is "N-neutral". As a software designer, I've learned that it's always better to solve general cases rather than specific ones, and it seems clear that "N = 1" is a special case that we should not be focussing on, to the exclusion of other values of N.

The current AMQP designs focus exclusively on "N = 1", indeed take this value of N as a universal constant. The AMQP work on reliability assumes "N = 1". I've shown that "N = 1" is a limited special case that addresses neither the needs of large networks, nor of high performance. We (iMatix) have no cases of "N = 1" deployments. So AMQP's reliability model, if it ever sees the light of day, will be useless for every one of the real deployments we have seen.

This should be enough to convince but I'll present one more reason why I believe reliability should not be built into AMQP's basic protocols.

Reliability is not one thing. The kind of reliability we want in a particular case is closely tied to the kind of work being done. We must start any design discussion for reliability with a clear statement of what kinds of messaging scenarios we are considering. Messaging is not one model, it is several, and each has cost-benefit tradeoffs, and each has a specific view of what "reliable" means. To give a non-exhaustive list:

- The request-response model, used to build service-oriented architectures. A caller sends a request, which is routed to a service, which does some work and returns a response. The simplest proven reliability model is a retry mechanism combined with the ability at the service side to detect and properly handle duplicate requests.

ul>
The transient publish-subscribe model, used to distribute data to multiple clients. In this model, if data is lost, clients simply wait for fresh data to arrive. A good example would be video or voice streaming.

- The reliable publish-subscribe model, used when the cost of lost data is too high. In this model, clients acknowledge data using a low-volume reply back to the sender. If the sender needs to, it resends data. This is similar to TCP.

Each of these looks like a distinct reliability protocol, each with different semantics and different interoperability, which tells me that they need to be layered on top of AMQP, rather than solved within it. Trying to solve reliability within AMQP means either that it will only solve one of the several cases, or it will try to solve them all, and be over-complex.

Let's recap what's wrong with centralized reliability:

- It does not handle the case where there is no server in the middle.
- It does not handle the case where we federate servers together.
- It does not fit on top of an asynchronous message flow.
- It wrongly assumes we need a single semantic model for reliability.

And circumstantially:

- It has proven approximately impossible to design.
- It has broken AMQP in many ways.
- There is no evidence it is the best model.
- There exist other designs that are simpler and proven.
- Other successful protocols don't try this.

Thus we come to the inevitable conclusions:

- The basic AMQP message transfer protocol should not have any semantics for reliability, acknowledgments, transactions, etc. It should imitate IP and be an optimistic, cynical protocol.
- Different types of reliability should be layered on top of this basic messaging protocol.

It is safe to assume that different types of security should also be layered on top of the basic messaging protocol. This is again how IP works and there has been no clear explanation of why that would not work for AMQP.

My conclusions will upset and annoy people who have spent a lot of time and money on trying to make the AMQP wire-level protocol implement perfect reliability. In my defense, I will say two things. First, this is not a new discussion. We explained it in early 2007 when we started our work on peer-to-peer messaging. Bringing the topic to a wider public seems necessary and overdue. Second, given the choice between annoying some people and getting a simpler package of protocols that has more chance of success, I'll choose the latter, any day. Life passes, but good protocols last forever.

What I'd really love to see are AMQP networks where boxes can fail without consequence, where data routes around damage, where excess load is handled by throwing stuff away, and where the full capacity of the network can be delivered to applications rather than consumed by heavy messaging architectures.

Once we accept that solving N-neutrality is more useful than solving "N = 1", the beginnings of a real architecture for AMQP start to emerge. There are hints of this in ZeroMQ, thanks to much careful design work by Martin Sustrik.

In the next section I'll propose what I believe should be the basic design for a new generation, rightly called AMQP/2.0.

A Simplified Model of Messaging

In this article on "What's wrong with AMQP (and how to fix it)", I've concluded that the current work on AMQP should be stopped, that the stable AMQP/0-9SP1 should be released as AMQP/1.0, and that a new approach is needed in order to properly solve AMQP/2.0.

My reasons for proposing this are partly that the current protocol workgroup is nearing burnout, and partly that I think the core architecture of AMQP has hit its limits and needs to be rethought. If my analysis is correct, all additions and refinements of the current core architecture are pure waste, and

will probably work against, not for, AMQP's long term interests.

There is no way, in my view, to gently evolve today's specifications into working ones. We need to lock-down the good stuff actually running in production, park everything else, and start once again from first principles.

This is what I'll aim to do in this article: start once again from first principles and build up to a new simplified model of messaging that can be the solid basis for AMQP/2.0.

Before general panic sets in, I'll say two things. First, to those who have already invested in AMQP, I've not questioned anything about how we use AMQP, only about how we make it. Implementations can, and must, change with new knowledge. This should be almost invisible to applications. It took us many attempts to build all the good, simple things in AMQP. Second, to those now yet familiar with the internals of AMQP, relax. The models I'll propose are simple. I'm aiming to make life easier for all of us, not harder. My goal is to get the core AMQP specification down to a few pages.

In AMQP, there is a primary elegant, powerful natural semantic: the exchange-binding-queue wiring model that gives application architects full control over how messages flow through the network.

Just a quick recap for those not intimate with this semantic. A queue holds messages, it is a FIFO buffer. An exchange routes messages, it is an algorithm, with no storage. A binding is a relationship between an exchange and a queue, it tells the exchange what queues expect what messages. Bindings are independent and orthogonal except that a queue will not get the same message more than once. There are two kinds of queue: private, for a single consuming application, and shared, for multiple consuming applications.

AMQP defines a set of exchange types each corresponding to a specific routing algorithm, and lets applications create and use exchange instances at runtime. It lets applications create queues at runtime, and bind them dynamically to zero or more exchanges. It then lets applications consume messages off their queue or queues.

The AMQP exchange-binding-queue semantic is a powerfully simple one and perhaps the most attractive feature of AMQP. However there are flaws in its design and understanding them is a good place to start making our new simplified model, which I'll develop by resolving those flaws one by one.

The first flaw is this: every message must always be routed via a queue in the server. This is, it turns out, redundant for the majority of use cases. How can server-side queues be redundant in most cases? Consider the three basic messaging scenarios:

- Point-to-point, where one application sends a message to another. The sender publishes a message to an exchange, which routes it to a private queue. The recipient consumes messages off its private queue and processes them.

- Data distribution, where one application sends a message to many others. The sender publishes a message to an exchange, which routes it to multiple private queues. The recipients each consume messages off their respective private queues, and process them.

- Workload distribution, where one application requests a service implemented by a group of applications. The sender publishes a message to an exchange, which routes it to a single shared queue. The recipients all consume off this shared queue, and get messages in a round-robin fashion.

What is missing from the above explanation is that each recipient has a further, hidden queue that is implemented by the client-side AMQP API. This hidden queue is invisible to the protocol but it is necessary because AMQP delivers messages to recipients asynchronously. AMQP does have mechanisms to send messages synchronously, but I believe these should be discarded - they originated in our reverse-engineering of JMS and they are clumsy and inconsistent.

In the point-to-point and data distribution scenarios, which are the heavy lifters in messaging, the server-side private queue, and the hidden client-side private queue do exactly the same work. It seems obvious when one starts with a JMS "N = 1" mindset, to hold private queues on a central server. But this design adds latency, puts stress on the server, and makes things more complex than they need to be by asking that applications create and manage extra entities.

Our first change to the AMQP model is therefore this:

- Client-side queues are addressable from the rest of the network. Specifically, exchanges can deliver messages directly into these queues.

What "addressable" means is that client-side queues can be used as the target for bindings, and that there is a protocol mechanism for delivering messages into those queues. We'll look at how this could work later.

The best counter-argument to the proposal that exchanges should be able to deliver directly into client-side queues is that any network activity needs a queue at both ends. I.e. if the network is slow, the exchange won't be able to deliver into a remote queue, and things will be blocked, so a queue is also needed at the sending side to buffer outgoing messages. Our experience is that TCP/IP buffering is adequate in most cases, and when networks really are saturated, the ideal response is to discard data, not create backlogs.

The counter-counter argument is that queues at the server side can always be added as a design choice for specific use cases.

Once we've turned the client-side queues into first-class protocol objects (i.e. addressable from the rest of the network), we can ask the question of how these are created and managed and named. The simplest sufficient answer, and thus the best one, is:

- Every client *always* has an queue for incoming messages. The lifespan of the queue matches the lifespan of the client. The queue has no name.

Let's now deconstruct the AMQP concept of "consumer". This is another JMS hangover and therefore on my hit list of concepts to get rid of. Consumers are poorly-defined relationships between server-side queues and message recipients.

Since client-side queues are now first-class objects, we can replace the fuzzy consumer concept with a more solid first-class relationship:

- Applications can bind queues to queues, as well as to exchanges. When an application binds its client-side queue to a server-side queue, this controls the flow of messages into the client-side queue.

For clarity, when I write "A binds to B", I'm using "binding" in the reflexive sense, as in "attaches to". A binding is an attachment made by a queue to a message source.

In other words, a queue-to-queue binding replaces the consumer concept and applications control the flow of messages into their client-side queues by creating and destroying such bindings.

Bindings now start to look very interesting: they are no longer just relationships between exchanges and queues but instead, connectors that can be stretched across the network. We start to get a glimpse of what AMQP's architecture could become.

If bindings are indeed elastic connectors that can stretch across the network, this lets us resolve another flaw in AMQP's model: that holding exchanges on a central server is a poor design for certain cases.

An exchange, doing vital work, must sit on a network node that has natural authority. This node cannot be allowed to fail, cannot be randomly stopped and started, cannot be moved without impact. In the point-to-point and workload distribution scenarios, no application node has natural authority, so we must create an artificial authority in the form of a central server. So far so good. But in the data distribution scenario, we have a good natural authority, namely the publisher. It makes no sense to create a second one. The redundancy makes things more complex, slower, and error-prone than they need to be.

If there is a chance to simplify things and make them faster and more reliable, we must seize it. So, we make the next change to AMQP's model:

- Exchanges may be anywhere on the network and are addressable from the rest of the network.

Since exchanges are just algorithms, they can presumably be built into client APIs just as well as into servers. There will be a cost, but in important cases this will be well worth paying. And in other cases, users can default to using server-side exchanges.

So with a few simplifications, AMQP now supports the optimal "N = 0" model for point-to-point and data distribution, while staying compatible with "N = 1" for workload distribution. How about "N > 1"? Mostly, federation does not appear to need any special protocol support; we do it neatly in OpenAMQ using the current semantics. Federation is, to remind the reader, done by attaching one exchange to another. It is not a binding, rather the client exchange acts as and simulates a consuming or producing application. However, we have changed the AMQP model so that private queues move off the centre and towards the edge, as client-side queues. This has an impact on federation that we must resolve.

Since the client exchange simulates an application, we put a queue in front of it so that it can receive messages from its parent exchange. Special cases should always raise a red flag. Does it make sense to put queues in front of some exchanges but not others? The simplest, most general answer is:

- Every exchange always has an queue for incoming messages. The lifespan of the queue matches the lifespan of the exchange. The queue has no name.

Which is neat because it makes exchanges look more like ordinary applications, and it lets make another major simplification:

- Messages are sent only, and always, to queues.

We have now suceeded in eliminating the old distinction between "publish" and "deliver", and we have also generalized queueing so that it always happens at the receiving side, which reduces delays, and every receiver has a first-class queue.

Message flow through an AMQP network now becomes fully N-neutral, its semantics don't change whether N is 0, 1, or higher. Some diagrams will show just how elegant this new model is. This is what the old "N = 1" AMQP model looked like:

```
           <==
[S] --> [X --> Q] --> [q --> R]
```

Where 'S' is the sender, 'X' is an exchange, 'Q' is a queue, 'q' is an invisible queue, and 'R' is the recipient. Each bracketed set represents a network node, and "—>" arrows represent message flows. The "<==" arrow is a binding.

Let's redraw this "N = 1" case using the new N-neutral semantics:

```
                  <==
[S] --> [Q --> X] --> [Q --> R]
```

Now, to build a fast peer-to-peer "N = 0" network we just re-arrange the brackets to get:

```
                <==
[S --> Q --> X] --> [Q --> R]
```

And we construct a wide-area "N = 2" federation as follows:

```
                  <==           <==
[S] --> [Q --> X] --> [Q --> X] --> [Q --> R]
```

Our workload distribution model uses a queue-to-queue binding:

```
                  <==    <==
[S] --> [Q --> X --> Q] --> [Q --> R]
```

It's clear that the N-neutral model is both simple and flexible enough to handle every messaging architecture we've come across.

Let me summarize what the new N-neutral model for AMQP looks like:

- Exchanges can be anywhere on the network.
- Every application has a automatic local queue for incoming messages.
- Every exchange has a automatic local queue for incoming messages.
- These queues are first-class objects, addressable from the rest of the network.
- It is still possible to hold exchanges and shared queues on a central server.
- Bindings can be from queue to exchange, and from queue to queue.
- Bindings can stretch across network links.
- The single semantic for message transport is "send message to queue".
- Exchanges and queues send messages according to bindings.
- All values of N are supported using the same semantics.

Based on this new simplified AMQP model, we can start to design real protocols, which is what I'll do in the next article.

Removed

The original text drifted into ad-hoc protocol design at this stage. We've removed that, it's not useful. However, if you take a close look at 0MQ, you will see that it does much of what we argue for in this article.