AMQ Background

Background to the AMQ Project

version 1.0a1

Pieter Hintjens

Revised: 2006/12/14

Contents

1 Cover

1.1 State of this Document

This document is a design white paper.

1.2 Copyright and License

1.3 Authors

This document was written by Pieter Hintjens in 2005-06, based on original texts and designs by John O'Hara of JP Morgan Chase & Co., and formed the basis for the AMQ Protocol Specifications. Many other people contributed ideas and suggestions that were included in part or whole in this document and the AMQ architectures.

1.4 Abstract

We explain the rationale behind the development of AMQ as an architecture and AMQP as a new industry standard protocol. This document is designed as useful background material for people wishing to understand the AMQP design.

2 Introduction

2.1 Defining Middleware

Wikipedia defines middleware thus: "In computing, middleware consists of software agents acting as an intermediary between different application components." (http://en.wikipedia.org/wiki/Middleware).

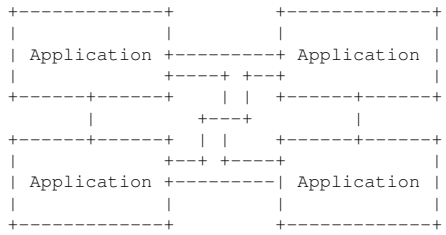
Wikipedia also comments that, "middleware is now used to describe database management systems, web servers, application servers, content management systems, and similar tools," and finally says that "the term is sometimes considered a buzzword."

This is somewhat like trying to define "the Web". The term means many things to different people. However, at the heart of it, "the web" comes down to the HTTP protocol, the HTML language, and functional servers such as Apache that bring it to life.

Similarly, "middleware" comes down to protocols, languages, APIs, and messaging servers. All the rest is layered on top.

2.2 Centralised Messaging Servers

The basic reason of being of a messaging server is to reduce the interconnection complexity of a network. In a network with no central messaging server, each application does its own message queueing and routing. We would use existing protocols such as SOAP to connect peers. If the whole network has to be reachable, each peer needs to connect to every other peer, which results in up to factorial(N-1) connections:

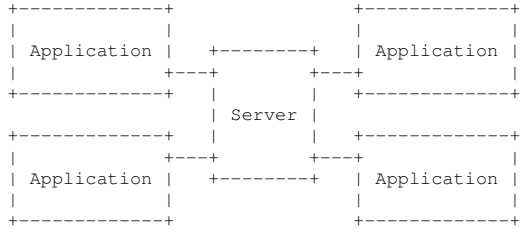


So, for four applications we need up to six connections, and for ten applications we need as many as 45 connections. Each connection is a project, meaning work, dedicated storage, probably a dedicated team, and each connection is an additional point of failure.

This is the traditional way of constructing ad-hoc application networks and it is a painfully unscalable model, as any examination of a large scale information network will show.

Peer-to-peer networks are one plausible solution but these are unproven for enterprise use. The simple issue of how to ensure reliability, i.e. safe storage of a message on a unbreakable disk, is not solved by using a peer-to-peer network.

A central messaging server moves the messaging complexity to one point, and then provides each application with a simple and unique way to access that complexity:



The general model for messaging servers, which has developed over some decades, is to provides disk-based or memory-based FIFO data stores (variously called queues, destinations, topics, and so on) that many applications can write to and read to simultaneously.

AMQ calls these data stores "message queues", and it is AMQ's prime task to provide applications with reliable, pervasive, fast, secure and economical shared access to a distributed network of message queues.

2.3 Existing Middleware Options

Middleware, in the form of messaging servers, is a platform on which software architects can create larger-scale applications by combining applications written using different technologies, running on different types of boxes. It is often significantly cheaper to develop independent applications and connect them using good middleware, than to develop a single all-encompassing application.

All working large-scale software projects necessarily use middleware of one kind or another. In most cases ad-hoc middleware is built for the needs of the project. Many commercial products also exist. Free products are also available. All these options entail significant costs.

Many projects build their own middleware. The problem often seems simple ("get the data from here to there, reliably and quickly") but the middleware aspects of projects often become the most complex and troublesome parts. People persist in writing their own middleware layers because using the off-the-shelf alternatives often means more work, not less.

Successful middleware products and technologies tend to occupy a specific niche: e.g. connecting components on a single platform, or for a single language environment. The few general-purpose middleware products (e.g. IBM MQ Series, BEA Tuxedo) are typically very expensive and very complex.

Lastly, there are some open source (mainly Java) middleware servers but these tend to be featureware, not focussed on standards, or rather, implementing so many different standards that there is no guarantee of interoperability even when using a single product.

There are some middleware standards (e.g. JMS, CORBA) but these are limited in scope. JMS, for instance, is exclusively for Java applications, though some JMS providers make non-standard APIs for other languages, and CORBA uses a complex object-centred model which is unsuitable for many kinds of application.

2.4 General Requirements

2.4.1 Economical

AMQ must be an economical solution. This means:

- Cheap or free to license.
- Available on all modern operating systems.
- Simple to install for default configurations.
- Self-managing and cheap to administrate.
- Easy and cheap to interface with applications.

2.4.2 Accessible

AMQ must be accessible to all users. This means:

- Low cost (see above).

- Compatible with applications built using any language, and any technology.
- Small footprint for devices with limited capacity.

2.4.3 Useful

AMQ must address realistic problems faced by enterprise applications. This means handling:

- Direct point-to-point communications, similar to a remote procedure call.
- Store-and-forward messaging, similar to email.
- Publish-and-subscribe messaging, similar to newsgroups.
- Multicasting, where messages are sent to many parties in parallel.

2.4.4 Quality of Service

AMQ must be able to sustain a service at or above certain minimum performance level. This includes:

- Handling large messages, limited only by the size of local file systems.
- Being able to guarantee maximum delivery times for specific services.
- Being able to guarantee minimum bandwidth for specific services.
- Being able to handle errors (such as undeliverable messages) and problems in a robust and useful manner.

2.4.5 Sophisticated

AMQ must be able, eventually, to handle realistically complex problems. This includes:

- Routing messages through a variety of means: by address, by content, by runtime calculation; to individual applications, groups of applications, temporary and permanent aliases, individuals within groups, and so on.
- Allowing multiple virtual networks (typically for development, test, and production environments) to exist within the same physical network.
- Ensuring the authentication of all parties, and encryption and signing of data when necessary.
- Allowing extension of the server to encode and transform data.

2.4.6 Enterprise Application Integration

EAI is a term that defines middleware that is able to act not just as a bridge to carry data between applications, but also to take control of the applications at either end. EAI middleware can act as a full-scale platform for application development (one of the reasons why commercial middleware is so complex). To be classed as EAI middleware, AMQ must be able to:

- Invoke processes to handle incoming data.
- Integrate with workflow management systems or provide workflow management facilities.

2.5 Conclusion and Strategy

We have these key goals and objectives:

1. To define an platform architecture that can become a standard for arbitrary implementations of middleware products.
2. To favor separation of layers so that hard problems can be solved independently and these solutions improved over time.
3. To define protocols between the different layers and components and promote these protocols as standards.
4. To sponsor high-quality reference implementations of the components.
5. To make these implementations pervasive and portable across all platforms, all applications, all domains.
6. To encourage community investment by adopting an open source license architecture that make it convenient and rewarding for skilled users to extend, modify, and improve these implementations over time.
7. To aim for deployment in real situations so that the product matures and proves itself quickly.
8. To provide a test suite to unambiguously qualify compatible software.

3 Functional Requirements

3.1 Messaging Models

We want to support a variety of messaging architectures:

- Store-and-forward with many writers and one reader.
- Transaction distribution with many writers and many readers.
- Publish-subscribe with many writers and many readers.
- Content-based routing with many writers and many readers.
- Queued file transfer with many writers and many readers.
- Point-to-point connection between two peers.
- Market data distribution with many sources and many readers.

Looking at each of these gives us a view of what we are aiming to achieve with AMQ.

3.2 Store-and-Forward

By definition, store and forward (S&F) defines a communications system in which messages are received, recorded, and then forwarded. We use the term in a more specific business context.

S&F is an end-to-end connection between applications. One application prepares messages to send to another application. It places these messages in a queue. The messages are delivered to the receiving application.

The S&F model has these characteristics:

- Typically the messages contain business data - new orders, database updates, client profiles, electronic documents.
- Messages are addressed to specific applications.
- Messages cannot be lost, once accepted by the middleware, and until delivered to the reader.
- Messages may be delivered under control of a transaction manager, or not, in which case we use the rule: deliver at least once, and reject duplicates.
- Messages are delivered in order. If there is one writer this means in order sent by the writer.
- Delivery synchronization is defined by the reader, signaling the middleware when it is ready to receive messages.
- Messages may be undeliverable, in which case they are placed in a dead-letter queue or returned to the publisher.
- Messages may be delayed, in which case the size of the queue will grow and possibly signal an alert condition.
- End-to-end transactions are the responsibility of the applications, not the middleware.
- There may be multiple writers to a queue but there is a single well-defined reader.

For store-and-forward applications we expect a response time of at most a few seconds under normal conditions. You would not expect an end-user to wait while a store-and-forward transaction finishes.

The volumes involved in store-and-forward applications can be up to 10-1,000 messages per second, with each message being 1-100Kb large. With store and forward the emphasis is on reliability and security rather

than on raw performance.

3.3 Transaction Distribution

In a transaction distribution architecture a message is a unit of work rather than a unit of information. A transaction distribution model is similar to S&F, with these changes:

- We enclose message delivery in a transaction so that errors in processing (not just delivery to the reader) can be communicated back to the queue.
- We allow multiple readers for a queue, and we define specific semantics for this, namely that messages are distributed to readers on a load-balancing basis (typically first-available round-robin).
- When a transaction is aborted, the message is re-queued and sent to a different reader. Thus message delivery is not guaranteed to be in order since the re-queuing may happen after other messages are sent from the queue.

Generally a transaction distribution model sends the message to a single reader but in some cases there may be two or more readers to assure redundancy (m of n distribution, deliver the message to m of n endpoints).

3.4 Publish-Subscribe

In a publish-subscribe (pub-sub) model, the sender does not send a message to a specific recipient but rather to a so-called topic, the "publish" action. Readers ask to receive messages on specific topics, which is the "subscribe" part.

Since a single message can be delivered to many readers, with the work being done by the messaging server, pub-sub is much more scalable than messaging queuing.

The pub-sub model has these characteristics:

- Routing is done using a topic name - readers subscribe by asking for specific topics or patterns that refer to multiple topics.
- Typically the messages expire or become less valuable quite rapidly.
- Messages may be held for a specific period only (typically seconds through to hours) after which they are purged.
- Message delivery is asynchronous and is usually not confirmed.
- There may be multiple writers to a topic.
- There are by definition multiple readers from a topic.

The response time for pub-sub systems used for event notification is often very important: market data expires rapidly. A stock price delayed by five minutes can be worse than worthless.

The volumes involved in pub-sub applications can be up to 100,000 messages per second, with each message being quite small (around 1-10Kb). With pub-sub the emphasis is on performance and specific qualities-of-service rather than reliability.

3.5 Content-Based Routing

Content-based routing (CBR) means that the recipient for a message is determined not by the sender but by the contents of the message. The server applies a set of rules on the message data, or calls an application procedure, to calculate the actual recipients.

CBR tends to be computationally expensive, so is used for more sophisticated applications where the volume of data is lower than in simpler pub-sub models. It is generally built and used as a layer on top of pub-sub.

3.6 Queued File Transfer

Occasionally, store-and-forward systems can be requested to transfer messages as large as or greater than several GB in size, for example holding backup sets. This can be thought of as unifying file transfer with enterprise messaging. These messages will take several seconds to store to disk prior to forwarding, so message rates are correspondingly low.

Queued file transfer is similar to S&F (one reader) or pub-sub (multiple readers), but:

- Messages are provided to the network as files (rather than memory blocks).
- Files can be sent in batches (I.E. "send any file in this folder").
- Files are delivered in the order received by the destination. If there is one writer this means in order sent by the writer.
- Files can be resent partially to recover from network failures.
- Received files can be restored if a transmission fails.
- No real limit on the message size: files can be as large as the file system allows.
- Files can be automatically deleted after sending.
- Text files can be converted automatically when sending between different systems.

3.7 Point-to-Point Connections

A point-to-point connection usually follows the model of a remote procedure call (RPC) although it can also be implemented as a peer-to-peer dialogue.

Remote procedure calls (RPC) are synchronous links between a client and a server application. From the point of view of the developer using the middleware, RPC is very different from store-forward or pub-sub. Internally it can be mapped to the same concepts, but with specific quality-of-service requirements.

RPC usually has visibility to the end-user of an application, so must complete rapidly. A typical RPC transaction can take up to 1/5th of a second before it starts to add visible delay for the end-user.

The RPC model has these characteristics:

- Typically the messages contain units of work - new orders, database updates, client profiles, electronic documents.
- Messages are addressed to specific applications.
- If we model traditional RPC systems, we allow the middleware to be unreliable and lose messages, assuming that the user or client application handles these errors. RPC systems do not hide much complexity from the developer.
- We may alternatively model a more sophisticated RPC system that provides reliable message transfer.
- Transactions are based on a request to a service and one or more responses back to the original sender.
- If they are undeliverable within a specific time-frame the sender must be informed directly.

The volumes involved in RPC applications can be up to 10-50 messages per second, with each message being 10-100Kb large. With RPC the emphasis is on quality-of-service rather than capacity.

RPC developers face additional problems that are less obvious to other middleware users. They must convert data into forms acceptable to (possibly foreign) applications, and they must manage large numbers of formalized message structures. Most RPC platforms provide toolkits to help with this work.

3.8 Market Data

The particular feature of market data is the huge volume of transactions. This has consequences on the implementation of any platform that supports it. We need to use technologies such as multicast or clustering to get a usable level of performance. AMQ does not define such functionality directly but it must allow it as an optional feature of an implementation.

4 What is AMQ?

4.1 Doing Better

AMQ must do better than the existing middleware solutions. We will look at the main deficiencies of today's products, and see how solving these drives the design of AMQ.

4.1.1 Interoperabilty

Interoperability means that when you plug a box from vendor A into a network with a box from vendor B, the two can talk to each other and do useful work. Today's middleware products are not interoperable except in the most basic sense: they will all speak TCP/IP to each other but not much more.

We can do better by standardising three distinct layers - the wire-level protocol, the semantics of the command set, and the APIs. For the APIs we can provide and use several alternatives since different applications have different requirements. But for the wire-level protocol and command semantics, we need absolute conformity in order to create functional interoperability. The commands to send messages, read messages, start and commit transactions must look the same, and must work the same.

4.1.2 Performance

Middleware is always a bottleneck, and can never be too fast. We want to be able to handle at least 10 times, and up to 100 times the volume of existing middleware servers. This requirement places constraints on many areas of the design.

The key to making a "fast" protocol (which enables but does not guarantee fast software) is to make a protocol that can be read and written in large chunks (not character by character), that uses a minimum of network capacity (since every octet costs time), and does a minimum of chatter (since every round trip costs a serious amount of time). 90% of the performance comes from 10% of the protocol, mainly the commands that transfer messages between server and client, so these commands must be carefully designed and be able to work with the state of the art in network technology (such as remote DMA).

4.1.3 Cost

Existing middleware systems are some of the most costly components in the software landscape. License fees are high. Expertise is expensive, since many different technologies mean there is no commodity market in experts. Middleware systems are often slow and need dedicated servers, which raises infrastructure costs. Middleware is often closed-source and inflexible, meaning that applications must work around deficiencies, which is expensive.

We can do better by making AMQ an open source technology, with reference implementations that run cheaply on arbitrary material.

4.1.4 Flexibilty

Flexible technology adapts to the problem rather than forcing the problem to be twisted to adopt the technology. Most existing products are designed to solve one problem, and they often do this very well, but since they do not interoperate and are closed, there is no scope for solving non-standard problems.

We can do better by aiming higher and making general-purpose designs that solve a wider set of problems, with no dependencies on specific operating systems, programming languages, or other transient technologies.

4.1.5 Complexity

This is probably the key problem with most middleware systems (and most technology in general). It is a truism that complexity is much easier than simplicity. It is easy and cheap to make complex systems. It is difficult and expensive to make simple ones that do the same work. There is a huge gap between the application's high-level needs (which in the middleware domain usually boil down just to "get next message" and "write message") and the technical implementation. For any given functionality, such as delivering a message to an application, there are numerous alternative techniques, each implying different tradeoffs between speed and reliability. The cheap but inadequate approach is to expose all this complexity to the application programmer, saying, "you choose".

We can do better by hiding complexity whenever possible, so that the application programmer sees a simpler world in which the best choices are already made, and things work as they should, out of the box.

4.1.6 Pervasiveness

The virtuous circle with technology is that the cheaper and simpler it becomes, the more people use it, and the cheaper and simpler it becomes. Even when the internal complexity (in terms of number of pieces and the problems they solve) of an operating system increases by several orders of magnitude over a decade, the external complexity (in terms of skill needed to install and use) falls until it reaches near-zero.

Middleware is still stuck at the slow-complex-costly phase and we want this to change. We want to make a technology that everyone can use. This means the technology must be simple to understand, easy to use, and it must work well on arbitrary operating systems, with arbitrary programming languages. Our goal is that whenever and wherever a programmer reaches for the tool marked "Talk to another application", they can use AMQ and get a solution that works rapidly and lasts forever.

4.1.7 Openness

No patents, no restrictive licenses, no proprietary software. These are basic rules for a commodity product. We cannot and do not know with a certainty that there are no patents which AMQ infringes on. However, all the design elements are either obvious, or based on existing prior art.

4.2 Email as a Model

Among the many networking protocols, the closest parallel to middleware would be SMTP. Protocols like FTP, HTTP, and SOAP assume each party is on-line at the same time and that the network is well-known in advance. Email assumes that the network is fragile, changing, and complex, which is more accurate in most cases.

The significant differences between email and the other protocols mentioned are asynchrony and persistence. Asynchrony means that messages are pushed towards their recipients. Persistence means that data is stored safely until everyone who needs it has finished using it.

If we start with email (the combination of SMTP, POP3, IMAP and other protocols) as a basic model and make some changes, we can start to see what a commodity middleware product could look like:

- Allow messages of any size to be transmitted (email has limits, usually several megabytes).
- Allow messages to be routed to many parties, according to rules that are defined at runtime.
- Allow messages to be routed by content as well as by address, something that is possible in email using filters at the receiving end.
- Allow the quality of the service between peers to be defined and measured.
- Provide a guarantee that a message is delivered, even if services fail intermittently on the way.

- Ensure data security and confidentiality, transparently to the end users.
- Support very high rates of communication, on the order of 10-100k messages per second between a set of applications.
- Provide the ability to deliver the data in a variety of ways: not just to a mailbox (like email) but also directly to applications via various call methods.
- Provide the ability to add value in the form of services that do useful things with messages.

4.3 Lessons from Linux

We know that AMQ is a large product that will take time to grow into a mature form. The Linux kernel and its distributions are a good example of how a small but well-designed core can become the basis for large and complex products through the gradual accretion of add-ons and extensions over time. We can achieve this highly-desirable goal by:

1. Defining an architecture that is both simple and general (defining the kinds of pieces we want to make).
2. Defining protocols that are also both simple and general (defining the way these pieces talk to one another).
3. Building a strong toolset that lets us (and others) build such pieces and extend the product.
4. Using an open source framework to encourage investment from the user community.

4.4 Design Assumptions

4.4.1 Network Visibility

We assume that clients can address servers directly. We do not assume that servers can address clients.

4.4.2 End-User Applications

Although we may support some common API models, we make no assumptions about the programming languages, technologies, platforms, or operating systems used to run end-user applications.

4.4.3 Fault Intolerance and Reliability

We make some assumptions about how far we can rely on different parts of the infrastructure to correctly do their work:

- We assume that the network transport layers deliver data either accurately (with no corruption or partial loss), or not at all. We call this property "reliability".
- We assume that local disk storage is unreliable: that data written to local filesystems is subject to possible damage, loss and corruption.
- We assume that any resource - network, destinations, and local filesystems - may be unavailable for indeterminate periods.

If the architect of a middleware network needs reliability at a certain point, he achieves this by investing in storage-area networks, server clusters, fault-tolerant servers, redundant network links, and so on.

We compensate for resources that are temporarily unavailable (usually by waiting) but we do not explicitly provide functions for fail-over. We provide some mechanisms that can be used for fail-over and load-balancing but the use of these in clustering scenarios falls outside the scope of this document.

4.5 AMQ Concepts

The key concepts that the AMQ design introduces:

1. A general-purpose modular framework for the server semantics (the AMQ framework).
2. A general-purpose wire-level protocol (AMQP).

4.5.1 The AMQ Framework

The traditional pre-AMQ middleware world has a large rift between its two main technologies: store-and-forward (S&F) and pub-sub (pub-sub). There are many products that do one or the other, but few products that do both well, mainly because there are inherent conflicts of interest - S&F focusses on reliability and pub-sub focusses on speed.

There have been moves to weld these two domains together, so Sun's JMS standard API has evolved from defining S&F and pub-sub as two different classes, to defining these as a single class under the broader term, "destination". The problem with bluring the distinction between S&F and pub-sub is that they have fundamentally different semantics, and the JMS standard is probably harder to use, not easier, because it attempts to mask these differences. JMS brokers tend to be neither particularly fast, nor particularly reliable.

The first prototypes of our reference implementation implemented S&F and pub-sub following the standard model, with two different engines grouped under a single "destination" concept.

It became apparent after much analysis that:

1. The "destination" concept is semantically too confused to act as a basis for a server design, and by extension, protocol design. Over time, we have completely removed this from the AMQ specifications.
2. S&F and pub-sub represent not two fundamentally different domains, but two special cases of a much wider spectrum. We were able to break S&F and pub-sub into a component grammar that lets us rebuild these two ways of working, but also construct new semantics.

The AMQ component grammar, which we call the "AMQ framework", is a language that lets architects create arbitrary middleware engines. This sounds complex but only has three basic components:

- A component (the message queue) that holds messages in a FIFO queue.
- A component (the exchange) that routes messages into message queues.
- A component (the binding) that defines the routing paths.

S&F consists of a simple exchange and sophisticated message queues. pub-sub consists of a sophisticated exchange and simple message queues. AMQ lets us make all other combinations as well.

4.5.2 The Wire-Level Protocol

Designing a wire-level protocol means solving a number of issues. Mainly, we wanted a fast and compact syntax capable of handling a complex set of commands and data.

The final design consists of a number of pieces, being principally:

- A general framing format.
- A general way of carrying commands.
- A general way of carrying data.
- A general way of handling errors.

The overall strategy has been to make general-purpose designs rather than specific ones. Each design solves one level of problem, and we work up from the basic wire level ("how do I decide how many octets to read next?") to the high functional level ("what does a distributed transaction mean?") in clear and fully independent layers, so that each design can be studied, prototyped, and improved by itself without affecting the rest. The segregation in AMQP is very strong, more so than in most existing protocols.

The risk of this strategy is that it makes AMQP harder to understand at the beginning. However, once the reader grasps the different layers, it is easier to understand each layer. The advantage is that we can extend and improve the protocol almost without restriction, cheaply and safely.

This is particularly important, since once third parties start to implement a protocol, it is near-impossible to make radical improvements to it. This is why HTTP still does not implement obvious improvements such as multiplexing.

5 Designing the AMQ Framework

5.1 Analysis of Problem

From analysing the main functional areas (S&F, transaction distribution, pub-sub, CBR, queued file-transfer, point-to-point, market data), we can see that the differences lie on these axis:

1. The size of the message, which ranges from very small to very large.
2. The type of the message, which includes application data, files, and possibly other types such as streamed data.
3. The reliability requirement, which ranges from low to very high.
4. The latency requirement, which ranges from unimportant to critical.
5. The routing model, which covers several different models: single named recipient, fanout, name-based routing, content-based routing, etc.
6. The distribution model, which covers several different models: single reader, message fanout, and workload distribution.

We must make a fundamental choice when we try to design a solution. Either we select some commonly-used combinations (some or all of the main functional areas), and design monolithic but dedicated solutions to each of these, or we find a way to modularise the problem into components that can be combined to create the common, and less common, usage patterns.

Our conclusion, after some detailed prototyping, was that there is a significant overlap between all these areas, and that the monolithic approach was over-complex and wasteful.

The key insight is that pub-sub and S&F, the two most extreme cases, share the same fundamental queuing and routing requirements. That is, the ability to store and deliver messages in the right way (where "right" has many shades of meaning), and the ability to route messages into the appropriate queues.

In pub-sub, the message queue is often hidden under the guise of "subscription", but when we see such real applications that need persistent, shared subscriptions, it becomes clear that these are just message queues, with almost an identical range of capabilities as S&F queues.

As for routing messages into message queues, the only real difference between the S&F model and the pub-sub model is that the latter is not directly coupled, i.e. the publisher does not know who the recipient is for a specific message. In the pub-sub model, consumers subscribe to topics, and publishers send to topics, and some matching engine does the hard work of matching messages with consumers and their individual message queues. If we take a little distance we can see that:

- All routing can be loosely-coupled, even S&F. There is no advantage to implementing the special case of "send message to queue X" when "send message to all queues that have asked for messages for X" has exactly the same semantics and performance, but is much more general.
- All matching can be done by specialised engines. The algorithm for doing high-speed topic matching is quite specific and has no place in S&F messaging.

What we need is some semi-formal definition of the data we match on, and some way to wire an arbitrary set of matching and routing engines with a set of queues.

Our terminology is this:

- The "routing key" is the main element we match on. This is like the "To" field of an email message. All messages have a routing key.

- The matching and routing engine is called an "exchange". We tried various other terms, including "router", but these all have invalid connotations. The exchange accepts messages, examines them, and routes them to a set of message queues as necessary.
- The tie between message queue and exchange is called a "binding". A binding is a specification, telling the exchange what messages to route into the queue.

5.2 The Exchange Concept

The "exchange" is an AMQ concept that abstracts a specific message routing mechanism.

Overall, an exchange is a logic engine that accepts messages, inspects them, and on the basis of pre-declared routing tables, routes the messages to a set of message queues.

The message queues are internal to the same server process. We might route messages between processes, especially in a clustering model, but this is not a primary functionalty of exchanges.

Exchanges have these properties:

- They do not store messages.
- They have arbitrary routing algorithms that use arbitrary criteria. We classify these different algorithms as "exchange types".
- They can duplicate messages, so a single input message can be routed to many queues simultaneously. The message may be held once, with reference counting, or actually duplicated. Each message queue has an independent instance of the message.
- They can produce or consume messages. That is, exchanges can be end-points for messages, and can also create their own messages when needed.

5.2.1 Types of Exchange

Looking at our known functional requirements, we need several types of exchange:

1. An exchange that routes on the routing key. This implements S&F transaction distribution, queued file-transfer, point-to-point.
2. An exchange that routes unconditionally. This implements fanout.
3. An exchange that routes on a routing key pattern. This implements pub-sub.
4. An exchange that routes on message header fields. This implements market data.
5. Arbitrary exchanges that route on the message contents. These implement content-based routing, message transformation, etc.

For basic interoperability we can standardise the first three or four of these exchanges. For extensibility we can let server implementations add their own exchange types, possibly as dynamically-loadable modules.

5.2.2 Exchange Instances

It rapidly becomes obvious that an exchange type is an algorithm, not an object instance. What is an exchange instance? It is the routing tables and binding information.

At the very least we need an instance of each exchange that we use, for each virtual host. We cannot allow message queues in different virtual hosts to be visible in the same space.

There is little reason to restrict this to one instance of each type per virtual host, and some advantages of allowing multiple instances:

- Scalability - exchanges may be scalable to certain limits, and to avoid passing those limits, it may be useful to split a heavily used exchange into two instances.
- Distribution - it may be useful to run separate exchanges in their own threads to better use a multi-processor system.
- Security - we plan to apply access controls to the data entering and leaving an exchange. This would require one exchange per access control realm.

5.2.3 Examples of Bindings

If we are routing on the routing key, a binding looks like this, using a simple pseudo-code:

```
SELECT INTO my-queue WHERE routing-key = "some value"
```

If we are routing on a routing key pattern, a binding looks like this:

```
SELECT INTO my-queue WHERE routing-key LIKE "some value"
```

where 'LIKE' implements whatever pattern matching we need.

5.2.4 Wiring Process

The full logic for creating and using a queue is:

- Create the queue
- Bind the queue to 1 or more exchanges
- Consume from queue

We have decided to make this entirely accessible from the protocol so that applications do not need to be configured before they can run.

The extra step, "bind the queue to an exchange" is troublesome for users that are still learning how the AMQ semantics work. We have found that using intelligent defaults makes this much simpler:

- First, the blank exchange name refers to an exchange instance that matches on the routing key.
- Second, all message queues are bound to this "default" exchange, using their queue name as routing key.

This means that the logic of creating and using a queue is reduced to:

- Create the queue
- Bind the queue to zero or more exchanges
- Consume from queue

Which is much easier, since for all directly-addressed messaging, no explicit binding step is necessary.

5.2.5 Mandatory Routing

Even when routing is loosely-linked, we need ways for applications to know whether or not a message was deliverable. There are several levels at which a message can be considered "undeliverable". At the exchange level, this is when the exchange has no message queues to which to route the message.

There are two distinct strategies for the exchange:

1. Attempt to route the message, and if there are no queues for it, drop the message.

2. Attempt to route the message, and if there are no queues for it, signal the application that there is a problem.

We call this "mandatory routing" and present it to the application as an option that it can set when publishing a message.

It is easy to signal an error when we have a synchronous protocol - the publisher says, "publish this", and the exchange replies, "OK", or "Failed".

But AMQP is asynchronous for all performance-critical commands, and this very specifically include publishing messages. We do not confirm success, because this turns the protocol into a bottleneck. The only way for an application to get success/failure status for an asynchronous command is to wrap it in a transaction, which creates a synchronous envelope. This makes things slow again. It is also plausible for applications to send "batches" of commands, so the server can accept or reject each one. This is complex and puts the burden on the application to manage its state.

Our solution is simpler and, we hope, more elegant. Asynchronous commands do not need any reply so long as they succeed. Failures can be signalled in several ways:

1. By explicitly returning the failed command, and saying, "this did not work".
2. By stopping the session (thread of control), saying, "something is wrong and needs to be fixed before you can continue".
3. By stopping the whole connection (and multiple sessions), saying, "you are doing something wrong, please get checked."

Since errors are (in a healthy case) very rare, we can afford to treat them as exceptional. Better, by making errors insurmountable, we avoid dangerous half-working, half-broken states. Either an application works, or it does not.

Back to mandatory routing. When a message cannot be routed, we return it to the application. We must return the whole message since we cannot know what the application needs in order to process returns. Returning just a message id assume that the application holds onto published data. Returning just message properties assumes that the application does not put anything important in the body. So, we return the entire message.

5.3 The Message Queue Concept

The "message queue" is an AMQ concept that abstracts the FIFO message stores that form the heart of most messaging servers. What we have done is to make this a very explicit concept that is visible to end-user applications, and which they can manipulate in interesting and useful ways. The reason for this is our observation that the architects of a middleware network (not necessarily the application developers) appreciate explicit control over their resources. It is like allowing a database developer to create indexes and temporary tables dynamically rather than forcing these to be created by a database administrator.

5.3.1 Private vs. Shared

The two main types of queue are private queues and shared queues. This can be generalised as follows: a message queue can handle N consumers, where N is limited to 1 for certain types of work, and can be any value for other types of work.

When a message queue is shared by multiple consumers, its messages are distribute amongst these consumers. When a message queue is private to one consumer, its messages are sent only to that consumer.

In a load-balancing clustering scenario, message queues must be on the same server that all their consumers

have connected to. For private queues this is trivial and obvious. For shared queues, less so. AMQP does not explicitly support clustering but does provide mechanisms to let applications know what server in a cluster is responsible for handling shared queues.

5.3.2 Durable vs. Temporary

A durable message queue is like a configured object: when the server (re)starts, the message queue is present and active. Temporary queues are destroyed when the server shuts down.

5.3.3 Automatic Deletion

It is very useful to let applications create and destroy message queues on the fly, since this lets us implement an elegant service-oriented network - each message queue acts as a service, and any number of applications can consume from the message queue as service providers.

However, applications do not know when they are the "last" application to use a message queue, so cannot delete it explicitly. To avoid old message queues polluting a server, we provide an auto-delete function whereby when the last service provider has stopped working with a message queue, the server deletes the queue (after a polite pause in case the service provider just went away temporarily).

5.3.4 Immediate Delivery

When we use service queues as described above, we need a way to tell the requesting application, "your message could not be processed". This can happen after some delay. We use the same return method as for mandatory routing - the message is sent back to the client, asynchronously, with a failure message.

5.3.5 Implementation-Specific Properties

Message queues can have quite specific properties that depend on the server implementation. For example:

- Maximum size of queue.
- What happens to new messages when the queue reaches its limit - are the messages returned, dropped, saved to persistent storage, or sent to a dead-letter queue?
- Are messages forced to be persistent?
- What storage device is the queue on, if persistent?

And so on. These do not affect interoperability, so we do not expose these properties in the protocol. However, we want applications to be able to create queues dynamically, with all possible properties.

Our solution is to use field tables - that is name + value pairs - in the queue declare method, so that these parameters can be specified without forcing them to be standardised in the protocol.

5.4 The Content Class Concept

The "content class" is an AMQ concept that splits the semantics of different functional domains in a clean way. A content class is like an object-oriented class (without the inheritence), and consists of a set of property definitions plus a set of methods.

Each message is an instance of a content class, which is why we use the term "content" in place of "message" in some areas of this document.

The three content classes AMQP defines as standard (a server may implement any or all of these) are:

- Basic content, for the standard messaging domain.

- File content, for queued file transfer.
- Stream content, for streamed data (video, voice, or other data).

AMQP does not provide methods for creating or destroying contents - this is specific to each API - but it does provide methods for transferring contents to and from the server.

The exchanges and message queues are able to handle any content type. This means AMQ provides the same routing and queuing mechanisms for streamed data, and large files, as it does for typical messages. It may be that the implementation of message queues is polymorphic and different for each content type. We also have different consumer semantics for each content type - streaming video needs different delivery semantics than queued files.

The content class concept makes it easy to add large new segments of functionality to the protocol, and to server implementations, without breaking or changing existing models.

5.5 Reliability vs. Performance

In general the trade-off between message reliability and speed is that the more reliable the system, the slower it will run. Reliability can be broken into several aspects:

1. The type of memory used: system RAM or disk storage. If we want messages to survive a system reboot, they must be on disk.
2. The degree of redundancy: none, mirrored. This is called 1-safe, 2-safe, etc. If we want to survive a system failure (e.g. disk crash) we need redundancy.
3. The redundancy distance: if we want a redundant system to survive a physical event (fire, lightning, hurricane), it must be physically separated from its peer.
4. The degree of transactionality: none, partial, full. If we want to be sure that the data was safely written in a coherent manner we must use transactions (of varying complexity).

In each case, there is a well-understood cost for the extra reliability. We wish these costs to be visible to, and under control of, the end-user application (when the choice can be made at runtime) or the middleware architect (when it is a configuration choice).

6 Designing the Wire-Level Protocol

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

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

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

6.1 The Protocol Header

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

The protocol header mechanism works like this:

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

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

6.2 The Framing Layer

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

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

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

- The frame header holds a frame type, letting us separate different types of frame at the most basic level. For instance, commands (we call these "methods") have a different frame type from data (we call this "content").

- The frame header holds a channel number, which lets the peers send frames for multiple channels across a single connection.
- The frame end octet lets us trap malformed frames, a typical problem with newly-written client code.

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

6.3 The Data Types

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

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

Everything that AMQP sends is composed of these data types.

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

6.3.1 Bit Data Type

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

6.3.2 Number Data Types

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

6.3.3 String Data Types

Strings are either:

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

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

6.3.4 Field Table Data Type

The field table data type is encoded as a long string but internally it holds a series of fields, specified as name, type, and value. Field tables hold data at the application level - e.g. message headers - so these fields

have slightly different types from simple fields. Integers are signed, there is support for decimal values (e.g. to hold exchange rates).

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

6.4 The Method Layer

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

We wanted the method layer to have these characteristics:

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

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

AMQP has features specifically designed for asynchronicity:

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

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

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

For each class we define a set of methods that:

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

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

6.5 The Content Layer

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

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

Our design is this:

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

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

6.6 The Error Handling Model

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

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

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

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

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

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

The advantages of this model are that:

1. Success is silent, in the case of asynchronous methods, so the protocol is not chatty, and fast.

2. Failure is unrecoverable, so errors get fixed. I.e. any fault is highlighted very rapidly, and must be fixed.

The technique of "clear failure" is widely used to force applications to be robust. This is part of the reasoning behind the use of assertions in programming. The goal in AMQP is to ensure that when a middleware architecture has been tested and appears to work, there really are no errors, grey areas, or corner cases that are being covered up by error recovery in the application.

6.7 Intended Operating Lifespan

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

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

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

1. Message sizes: the largest messages (files) are today around 20GB. We expect this to grow by 50% per year, reaching the 16Eb simple-content limit in 50 years, and the 1Yb multipart content limit in 65 years.
2. Frame sizes: IPv4 is limited to 64KB frames, IPv6 to 4GB frames. Ethernet itself is limited to 12000 byte frames due to its CRC algorithm. We expect the maximum networking frame size to grow by 50% per year, in large leaps. We will thus reach the limit of 64-bit sized packets in 50 years.
3. Protocol classes and methods: the current protocol defines about ten classes and about ten or fewer methods per class. We assume that backwards compatability will be maintained by defining new classes and methods rather than modifying existing ones that are in use. The limit of 64k classes and methods per class should be sufficient to last more than 50 years.
4. Channels: the limit of 4G channels allows growth of 50% per year from an estimated usage of 10 channels per connection today.
5. Timestamps: the classic 31-bit time_t value rolls over in 2038, and the 32-bit extension on 2106. We use a 64-bit timestamp which is valid until past the end of the Universe. (Editor's note: is this necessary, given that the 32-bit extension is already valid long past our target lifespan of 50 years?)

6.8 An Extensible Functionality

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

1. Adding new types of exchange to server implementations.
2. Adding new properties to content classs.
3. Adding new arguments to methods.
4. Adding new methods to classes.
5. Adding new content classes.
6. Adding new method classes.

7. Adding new frame types.
8. Adding new protocols.

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