# AMQ RFC003
# A Fast Dispatcher Module

version 0.1

iMatix Corporation <amq@imatix.com>

Copyright (c) 2004 JPMorgan

Revised: 2005/02/10

# Contents

# 1 Cover

## 1.1 State of this Document

This document is a request for comments. Distribution of this document is currently limited to iMatix and JPMorgan internal use.

This document is a provisional proposal. This document is ready for review.

## 1.2 Copyright Notice

This document is copyright (c) 2004 JPMorgan Inc.

## 1.3 Authors

This document was written by Pieter Hintjens <ph@imatix.com>.

## 1.4 Abstract

One of the key challenges in publish/subscribe messaging is to perform fast matching of messages with large numbers of subscription requests. We propose a design for an extensible and high-speed dispatcher for use in OpenAMQ drivers.

# 2 Introduction

## 2.1 Problem Statement

The core of a publish-subscribe messaging system is the distribution or routing of messages to interested parties. This involves comparing each incoming message with a set of requests made by different clients, and on that basis deciding whether or not each client will receive a copy of the message.

Messages are typically matched on a small number of fields, often placed in a predictable location in the message structure, e.g. as part of the message header. We typically match on specific values.

In general there are many ways of comparing messages with requests. We are interested in a specific case:

1. The format of messages and requests is fairly simple and predictable.
2. Matching can be done on well-defined "criteria".
3. The volume of traffic is high and matching can present a bottleneck.

## 2.2 Argumentation

The only argumentation for our proposal is this: matching large amounts of data requires more than a list of items we can scan through. We have built very fast searching engines before using the technique of inverted bitmaps. This works by pre-indexing data into a long bitmap for each term or criteria that we can search on. One bit represents one document, 1 if the term is present in that document, else 0. The bitmaps are compressed and stored. To search the entire document database for one or more terms we retrieve the corresponding bitmaps, perform logical OR or AND operations on them. The resulting bitmap is an index of the documents that match.

The key gain in this approach is that there are few database accesses (if the bitmaps are stored persistently), and that the core work is done in large chunks (matching 32 or 64 "hits" per iteration through the bitmaps.) The SFL modules sflbits, sflsearch, and sflscrit.

While the matching engine in OpenAMQ does not entirely match this model, it is our starting technology.

## 2.3 Basic Proposal

When we want to handle high volume, we must simplify the work that is done. There are many ways of doing matching but we must force a model that is as fast as possible, while leaving some scope open for implementation-defined intelligence.

Thus our design is based on these elements:

1. A general structure for representing match criteria.
2. A general model for parsing messages and requests into criteria.
3. A general model for comparing these criteria (and thus determining a "match")

4. A general model for allowing implementation-specific tuning of this process.

We use the concept of a server "module" (already used in OpenAMQ) to define such implementation-defined intelligence. A module is a piece of code that is linked statically or dynamically and provides some specific functionality, via a simple registration process.

Our proposal is based around two types of module: a dispatcher and a parser, a general implementation for criteria, and a dispatcher design aimed at our requirements for speed.

# 3 Design Proposal

## 3.1 Definitions and References

**Criteria** The unit of matching. A criteria is an object with these properties: name, value, and type (number or text).

See AMQ_RFC008 for additional terminology.

## 3.2 Objectives

1. To perform high-speed matching of messages, with typical volumes being 500 clients making 10,000 total requests.
2. To create a general-purpose matching engine that can be integrated into OpenAMQ for those cases (types of destination) where we need it.
3. To allow arbitrary extensions of the matching engine so that customised logic can easily be added.

## 3.3 Architecture

The general architecture uses an abstracted concept of a "criteria", being a named item with some value. Requests are pre-indexed according to the criteria they ask for, so that each request can be represented by a set of criteria. Messages are analysed to extract criteria, so that each incoming message can be represented by a set of criteria. A message is considered to match a request when the request criteria form a subset of the message criteria. A client may make multiple requests to create combinations.

Our architecture is based on these design elements:

1. A general model for analysing messages and requests into criteria.
2. A general model for comparing message and request criteria sets.
3. A general interface for presenting this logic to external layers.

## 3.4 Proof and Demonstration

This proposal will be demonstated by a working parser and dispatcher that use a standard test suite.

## 3.5 Detailed Proposal

### 3.5.1 The Dispatcher Interface

The matching engine is implemented as an OpenAMQ module of the amq_disp_if class. The amq_disp_if API provides a general way to add matching engines to OpenAMQ, our present case being a specific implemen-

tation.

The amq_disp_if API will be refined and detailed in a separate AMQ RFC. For now we sketch this API, which is based on a dispatcher object, being an independent context within which a matching engine does its work.

**new** Create a new dispatcher object. The dispatcher maps onto a destination or set of destinations at the behest of the driver. Requests and matches are localized within the dispatcher object.

**request** Add a new client request to the set of active requests held by the dispatcher object.

**cancel** Remove all requests for a specified client in the dispatcher object.

**match** Compare a message with all requests held by the dispatcher object and forward the message to each matching request.

**destroy** Destroy a dispatcher object and cancel all its requests.

Note that the definition of "request" and "message" is left deliberately vague: at this level these are simply binary data blocks who's structure and syntax is not known to the server. Similarly, we will keep the structure and syntax of the message and request blocks unknown to the matching engine.

### 3.5.2 The Parser Interface

The actual message and request parsing is implemented by an OpenAMQ module of the amq_pars_if class. The amq_pars_if class provides a general way to add message and request parsing intelligence to OpenAMQ. All matching engines are compatible with all parsers, the combination is chosen by the driver as part of the driver configuration.

In our reference implementations, the parser is identified by the MIME type of a message (note that all AMQ messages are tagged with a Content-Type indicator).

The parser interface will be refined and detailed in a separate AMQ RFC. For now we sketch this API:

**Parse request** Given a request, return a list of criteria.

**Parse message** Given a message, return a list of criteria.

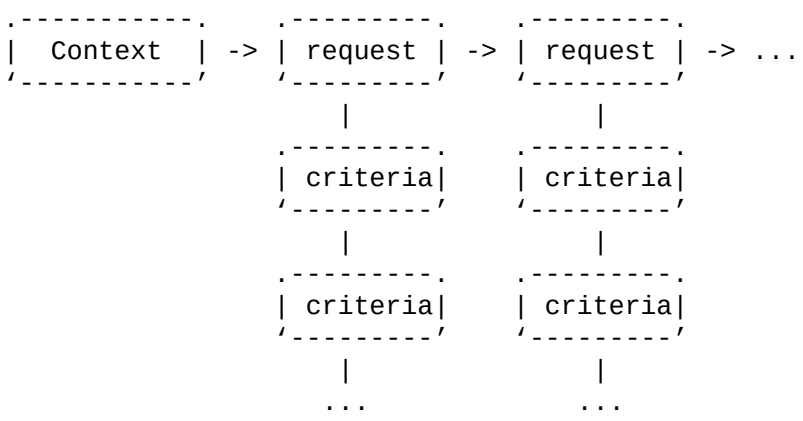
A criterion has these properties: a name (typically the name of a field in the message header or body), a type (number or text), and a value (the actual value of the field in the message).

### 3.5.3 Naive Implementation

Let's start by restating the problem: "Given a message, calculate a list of all clients who have made requests that correspond to the message. We defined 'correspondence' as matching on all requested criteria."

It's not a trivial problem. A client can make multiple requests. Each request asks for multiple criteria. A "match" is when a message matches all the criteria for one or more of the requests. We cannot mix criteria for different requests.

A simple implementation is as follows. We have a "context" object that holds a list of requests each holding a list of criteria for one client. We call this list the "request list" to keep things simple:




When we get a new request, we parse it and add it to the end of the list.

When we get a new message, we parse it, and we then compare it with each request. This means comparing each criteria in the message with each criteria in the request, and if they all match, we track the client reference.

When we have matched all the requests, we eliminate duplicate client requests, and return the resulting set.

It's easy to understand the above implementation, but it looks like it's doing a lot of work. Let's define the number of clients as "K", and the average number of requests per client as "R". The average number of criteria per request is "C". The average number of criteria per message is "M". When we scan the criteria list for a request, we can halt if we get any mismatches. To understand the average number of comparisons per request, we must know or estimate the total number of different request criteria (being the combination of name, value, and type). We will call this quantity "T".

The mathematics of the workload is left as an exercise to the reader. Note that M may be less than, or greater than T.

In a typical case K=500, R=4, C=5, M=10, T=200. Our average workload per message is about K*R, 2000 comparisons.

### 3.5.4 Improved Implementation

When we come to improve our model, we can use the observation that the cost of storing a request may be high: we do this much less frequently than matching a message.

Rather than store each client request as a separate entity, we can merge identical requests, and list the clients that have asked for each one.

In our worst case, every client request is different, and we still make K*R requests. In our best case, requests overlap significantly and the number of comparisons can approach T.

However, this is not good enough: we need a model where the worst case is still very fast.

### 3.5.5 Re-designed Implementation

We have already noted that we have about K*R requests to manage. Let us number these requests from 0 upwards. New requests are given a new number. When we purge requests we can re-use those numbers. We store the requests in a simple table so that we can perform a rapid lookup on a request, given its number,

and find the client that asked for it.

Storing a new request is an operation that can take some time: this happens long before the criticial processing loop of message matching.

Each request specifies a number of criteria. We invert this relationship so that rather than storing criteria per request, we store requests per criteria. Each criteria has a set of requests that ask for it. We can store this set as a bitmap, where each bit number corresponds to a request number. Our total number of criteria is T, so we have T such bitmaps.

We store these bitmaps in a hash table so that given a specific criterion, we can rapidly find its bitmap.

Our matching algorithm is now this:

1. Parse the message into a list of message criteria.
2. For each message criteria, lookup the bitmap representing the requests that ask for it.
3. Perform a logical AND on all of these bitmaps.
4. The final result is a bitmap of the requests that match.
5. For each remaining bit, lookup the client reference.
6. Collect the unique list of client references.

To perform the logical AND, we merge each bitmap into an accumulator using an assignment the first time, and a logical AND thereafter.

What is the workload of this algorithm? Merging bitmaps is very fast: if we hold 2000 bits (for typical values of K and R) as a simple bitstring, this represents 63 long words or 32 64-bit words. We can unroll the merge loop and approach one machine instruction per long word. Note that the average number of requests per criteria is T/R, so it is unlikely that we can use the sparseness of the array to any advantage.

### 3.5.6 Range Criteria

A request can ask for criteria with a certain range. For instance, where a value lies between two limits, is greater than some limit, or less than some limit.

The challenge with range criteria is to map the message criteria to all range criterias that have been requested.

Whether this is possible or not depends on the parser API. If the parser API is static: i.e. a message is always parsed in the same way, then we cannot map message criteria to request range criteria.

We resolve this by making the parser context-sensitive so that it knows all the range criteria that have been requested. An example may make this clearer:

- We have a request with the criteria "currency = USD, value = 51.2 to 55".
- We get a second request with the criteria "currency = USD, value = 53 to 60".
- This gives us three distinct criteria, and three bitmaps.
- We now receive a message with the criteria "currency = USD, value = 53".
- From this we derive three criteria: "currency = USD", "value = 51.2 to 55", and "value = 53 to 60".

- We lookup and AND the three bitmaps, after which we know which clients to forward the message to.

It is clear that adding this kind of intelligence to the parser makes it slower, and we will need to find fast mapping algorithms to map a message onto the correct criteria, so range criteria may be a feature we implement in a later stage.

## 3.6 Alternatives

We do not propose any alternatives at this moment. However, we provide a naive "brute-force" algorithm that can be used to test the APIs and the overall concepts.

## 3.7 Security Considerations

This proposal does not have any specific security considerations.

# 4 Comments on this Document

## 4.1 Jonathan Schultz, 2004/10/03

I don't know much about this stuff except at an abstract theoretical level, but, it would seem to be useful to think in terms of 'fuzzy logic' here. For phonetic matching, the reasons are obvious, but for text matching, we don't want just match/non-match, and even for numeric, it is rare that we want a strict cut-off rather than a fade-out.

Reply from PH: in the main cases, requests are not made by people but by programs, I.E. they are like SQL Select statements, written to extract specific data from a set. So we want exact matching on strings and strict cut-offs on numbers. I agree that when handling ad-hoc queries, we need different ways of defining a "match" but that is something for a later time. In any case, this is really the work of the parser, which is where we might do things like strip off suffixes. I will remove reference to "phonetic" in the text because it implies something we're not doing.