OpenAMQ Matching
Rapid Message Matching

version 1.0

iMatix Corporation

Revised: 2006/07/18

Contents

1 Cover

1.1 State of this Document

This document is a design white paper.

1.2 Copyright and License

1.3 Abstract

This document describes a matching algorithm that provides all the functionality needed for topic matching and most of the functionality needed for content-based routing.

2 Argumentation

Matching messages to requests is a typical bottleneck in a message oriented middleware server. The standard mechanism is the "selector", an SQL-like clause that is interpreted to give a true/false result. Selectors are the basic tool for "content-based routing". Given a high volume of messages and requests, the cost of selectors grows exponentially. The "topic" mechanism used by middleware servers provides a faster matching algorithm based on a hierarchical naming system, but this still acts as a bottleneck in high-volume scenarios.

2.1 Conventional Matching

We define a "request" as consisting of one or more "criteria", and a message as providing one or more "fields". The request specifies the desired criteria in terms of fields: for example, a field having a certain value, or matching a certain pattern.

Topic matching is based on patterns (e.g. "forex.*") while content-based routing is based on field values ("currency = USD or GBP").

The most obvious matching technique is to compare each message with each request. If the cost of such a comparison is C, then the cost of matching one message is:

```
C * R
```

Where R is the number of requests. The cost of C is proportional to the complexity of the request - i.e. the average number of criteria per request. In a low-volume scenario, R might be 1-10. In a high volume scenario we might have:

- 10,000 active requests (R = 10,000)
- a matching cost of 100 microseconds (C = 100)

Giving a cost of 10000 * 100 microseconds per message, or 1 second per message.

We can improve this by remarking that many requests are identical. If we assume that the maximum value for R will be around 100, we may reduce the cost per message to 0.01 seconds per message, giving us a maximum throughput of 100 messages per second.

Our goal is to get a matching cost of under 10 microseconds per message, for a potential throughput of as much as 100,000 messages per second.

2.2 The Inverted Bitmap Technique

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

Svalgaard and Tveden invented the concept of "inverted bitmaps" to enable rapid matching of requests with names in the directory.

The inverted bitmap technique is based on these principles:

1. We change data rarely, but we search frequently.
2. The number of possible searches is finite and can be represented by a large, sparse array of items against criteria, with a 1 in each position where an item matches a criteria and 0 elsewhere.

The indexing technique works as follows:

1. We number each searchable item from 0 upwards.

2. We analyse each item to give a set of "criteria" name and value tuples.

3. We store the criteria tuple in a table indexed by the name and value.

4. For each criteria tuple we store a long bitmap representing each item that matches it.

The searching technique works as follows:

1. We analyse the search request to give a set of criteria name and value tuples.

2. We look up each criteria tuple in the table, giving a set of bitmaps.

3. We AND or OR each bitmap to give a final result bitmap.

4. Each 1 bit in the result bitmap represents a matching item.

Note that the bitmaps can be huge, representing millions of items, and are usually highly compressible. Much of the art in using inverted bitmaps comes from:

1. Deriving accurate criteria tuples from items and search requests.

2. Careful compression techniques on the large sparse bitmaps.

3. Post-filtering search results to discard false positives.

Today's web search engines use such techniques. We (Hintjens et al) have built several search engines using these techniques (from STAR in 1990, to sms@ in 2001).

2.3   Application to Message Matching

The inverted bitmap technique thus works by pre-indexing a set of searchable items so that a search request can be resolved with a minimal number of operations.

It is efficient if and only if the set of searchable items is relatively stable with respect to the number of search requests. Otherwise the cost of re-indexing is excessive.

When we apply the inverted bitmap technique to message matching, we may be confused into thinking that the message is the "searchable item". This seems logical except that message matching requests are relatively stable with respect to messages.

So, we must invert the roles so that:

1. The "searchable item" is the matching request, which we will call a "subscription" for the purposes of discussion.

2. The "search request" is the message.

The indexing process now works as follows:

1. We number each match request from 0 upwards.

2. We analyse each match request to give a set of criteria tuples.

3. We store the criteria tuples in a table indexed by name and value.

4. For each criteria tuple we store a long bitmap representing each match request that asks for it.

The message matching process works as follows:

1. We analyse the message to give a set of criteria tuples.

2. We look up each tuple in the table, giving a set of bitmaps.

3. We accumulate the bitmaps to give a final result bitmap.

4. Each 1 bit in the result bitmap represents a matching subscription.

2.4 Worked Examples

2.4.1 Topic Matching Example

Imagine we have these topics:

```
forex
forex.gbp
forex.eur
forex.usd
trade
trade.usd
trade.jpy
```

Imagine these subscriptions, where * matches one topic name part and # matches one or more:

```
0 = "forex.*"
    matches: forex, forex.gbp, forex.eur, forex.usd
1 = "*.usd"
    matches: forex.usd, trade.usd
2 = "*.eur"
    matches: forex.eur
3 = "#"
    matches: forex, forex.gbp, forex.usd, trade, trade.usd, trade.jpy
```

When we index the matches for each subscription we get these bitmaps:

```
Criteria        0 1 2 3
----------------------
forex           1 0 0 1
forex.gbp       1 0 0 1
forex.eur       1 0 1 1
forex.usd       1 1 0 1
trade           0 0 0 1
trade.usd       0 1 0 1
trade.jpy       0 0 0 1
```

Now let us examine in detail what happens for a series of messages:

```
Message A -> "forex.eur"
    forex.eur  1 0 1 1 => Subscriptions 0, 2, 3
Message B -> "forex"
    forex      1 0 0 1 => Subscriptions 0, 3
Message C -> "trade.jpy"
    trade.jpy  0 0 0 1 => Subscription 3
```

The list of topics must either be known in advance, so that a subscription can be correctly mapped to the full set of topic names it represents, or the subscriptions must be "recompiled" when a new topic name is detected for the first time. In the OpenAMQ server implementation we take the second route, allowing topic names to be specified dynamically.

2.4.2 Field Matching Example

For our example we will allow matching on field value and/or presence. That is, a subscription can specify a precise value for a field or ask that the field be present.

Imagine these subscriptions:

```
Nbr  Criteria                    Number of criteria
---- --------------------------- ------------------
0    currency=USD, urgent        2
1    currency=EUR                1
2    market=forex, currency=EUR  2
3    urgent                      1
```

When we index the field name/value tuples for each subscription we get these bitmaps:

```
Criteria        0 1 2 3
----------------------
currency        0 0 0 0
currency=USD    1 0 0 0
currency=EUR    0 1 1 0
market          0 0 0 0
market=forex    0 0 1 0
urgent          1 0 0 1
```

Now let us examine in detail what happens for a series of messages:

```
Message A -> "currency=JPY, market=forex, slow"
    currency=JPY    0 0 0 0
    market=forex    0 0 1 0
    slow            0 0 0 0
    ----------------------
    Hits            0 0 1 0
Message B -> "currency=JPY, urgent"
    currency=JPY    0 0 0 0
    urgent          1 0 0 1
    ----------------------
    Hits            1 0 0 1
Message C -> "market=forex, currency=EUR"
    market=forex    0 0 1 0
    currency=EUR    0 1 1 0
    ----------------------
    Hits            0 1 2 0
```

Note that the hit count is:

- zero for subscriptions that do not match.
- 1 or greater for subscriptions that have at least one matching criteria (a logical OR match).
- Equal to the criteria count when ALL criteria match (a logical AND match).

2.5 Fast Bitmap Implementation

The bitmap implementation is a key to good performance. We use these assumptions when making our bitmap design:

1. The maximum number of distinct subscriptions is several thousand. The bitmap for one criteria can therefore be held in memory with no compression.

2. The maximum number of criteria is around a hundred thousand. Thus the bitmaps can be held in memory (on a suitable sized machine).

The bitmap engine provides methods for:

- Allocating a new empty bitmap.
- Setting or clearing a specific bit.
- Testing whether a specific bit is set to 1.
- Resetting a bitmap to all zeroes.
- Counting the number of bits set to 1.
- Performing a bitwise AND, OR, or XOR on two bitmaps.
- Performing a bitwise NOT on a bitmaps.
- Locating the first, last, next, or previous bit set to 1.
- Locating the first, last, next, or previous bit set to zero.
- Serialising a bitmap to and from a file stream.

2.6   Fast Matching Test Program

We explain a sample program that performs this matching process on a set of randomly-generated topics and messages. The program is provided in the OpenAMQ source package (kernel/amq_test_matching.c).

The core of the program does three things:

```
generate_topics        (nbr of topics);
generate_subscriptions (nbr of subscribers);
generate_messages      (nbr of messages);
```

We generate a set of topics built by selecting a random component from these tables:

```
char
    *topic_level1 [] = { "gold", "oil", "pork", "chips", "bonds", "debt" },
    *topic_level2 [] = { "lon", "ny", "hk", "jo", "tk", "sg", "mo" },
    *topic_level3 [] = { "usd", "eur", "gbp", "jpy", "chy", "aud" },
    *topic_level4 [] = { "buy", "sell", "hold", "short", "long" },
    *topic_level5 [] = { "spot", "early", "late", "open", "close", "future" };
```

We generate a set of subscriptions and for each subscription, match it against all known topics to build the inverted bitmap table.

We finally generate a set of messages with random topic names and perform the matching:

```
ipr_shortstr_t
    topic_name;
amq_match_t
    *match;
int
    count,
    subscr_nbr;                     /*  Subcriber matching topic       */
hit_count = 0;
for (count = 0; count < nbr_messages; count++) {
    /*  Our message consists simply of a topic name chosen at random   */
    strcpy (topic_name, topic_list [randomof (topic_count)]);
    /*  Find all subscribers for this topic                            */
    match = amq_match_search (match_topics, topic_name);
    if (match) {
        for (IPR_BITS_EACH (subscr_nbr, match->bits)) {
            /*  'publish' to subscriber, by incrementing hit counter    */
            subscr_table [subscr_nbr].hits = 0;
            hit_count++;
        }
    }
}
```

Real-time performance on a 2Ghz workstation is as follows:

- Building a list of 2000 subscribers and compiling for 2000 topics (4M comparisons) - 12 seconds.
- Generating 1M messages and matching against all subscriptions (170M matches) - 26 seconds.

Conclusion - we can do roughly 3M matches per second per Ghz.

2.7   Security Considerations

This proposal does not have any specific security considerations.