

An Efficient, Scalable Content-Based Messaging System

Ian Gorton, Justin Almquist, Nick Cramer, Jereme Haack, Mark Hoza

*Information Sciences and Engineering,
Pacific Northwest National Laboratory, Richland, WA 99352, USA
Ian.gorton@pnl.gov*

Abstract

Large-scale information processing environments must rapidly search through massive streams of raw data to locate useful information. These data streams contain textual and numeric data items, and may be highly structured or mostly freeform text. This project aims to create a high performance and scalable engine for locating relevant content in data streams. Based on the J2EE Java Messaging Service (JMS), the content-based messaging (CBM) engine provides highly efficient message formatting and filtering. This paper describes the design of the CBM engine, and presents empirical results that compare the performance with a standard JMS to demonstrate the performance improvements that are achieved.

1. Introduction

Finding relevant information in the ever-growing sources of digital information is a highly challenging research problem. At the core of this problem is the need to rapidly identify individual data items such as text documents, email messages and transaction data that are of interest from dynamic data streams containing ten's of millions of pieces of information each day. The sheer scale of these information sources, both in raw size and message arrival rate, makes efficient and cost-effective processing of the data highly problematic.

Processing continuous data streams [1] poses some unique problems. A streams processing environment must assume that the messages it receives are transient, and cannot be stored for rapid post-processing. Although some approaches attempt to offer rich query languages for long-running continuous queries on streams [2], these systems are still essentially research prototypes.

More limited query capabilities are embodied in technologies that are typically known as content-based messaging (CBM) systems. CBM systems employ some mechanism for querying the content of an individual message and delivering messages with the desired content to registered subscribers. Systems such as Elvin [3], Gryphon [4], XMLBlaster [5] and Siena [6] all achieve similar forms of content-based message notification on textual data arriving in message streams.

The querying capabilities of these systems do however vary considerably. For example, Elvin has a sophisticated subscription (query) language, whereby the evaluation of a subscription uses Lukasiewicz's tri-state logic, adding an *undecidable* value to *true* and *false*. Siena on the other hand offers a limited set of Boolean and string operations available for querying, and focuses on attempting to provide efficient query evaluation based on defining a *covering* relationship between queries [6].

Some commercial, standards-based technologies also can provide content-based messaging. Implementations of the CORBA Notification Service specification and Java Messaging Service (JMS) API include the ability to select or filter messages using a query language based on SQL-92. These technologies have multiple implementations, including open source, and are widely utilized and deployed in applications.

This paper describes the work in a project at PNNL that is attempting to design and build a CBM platform that has a rich message query mechanism and can scale to support massive volumes of messages in multiple data streams. In [7], Carzeniga *et al* cogently describe the inherent tensions between expressivity and scalability that exist in content-based messaging. Essentially, as the complexity and volume of queries grows, the cost of satisfying the queries grows significantly, and eventually leads the content-based matching engine to enter an unusable state. Our approach is based on enhancing the capabilities of the Java Messaging Service with algorithms for highly efficient and scalable message filtering. By leveraging standards-based technologies, we hope the path to adoption for the outcomes of the research can be shortened.

The paper describes briefly the message filtering capability of the JMS. The architecture of the proposed enhancements is then described, along with the improved algorithms for implementing message filtering in the JMS. The performance gains from these modifications are demonstrated through a set of test cases, which compare the throughput of the open source JBoss JMS with our enhanced version. The paper concludes by outlining our

plans for further work to significantly enhance JMS capability, performance and scalability.

2. Message Filtering in the JMS

The JMS is a mandatory part of the Java 2 Enterprise Edition (J2EE) platform. The JMS specification defines a set of Java interfaces and associated semantics for an asynchronous messaging system. Individual vendors implement the JMS by building a JMS *provider* that supports the JMS interfaces. This can be done by wrapping an existing messaging technology such as MQSeries, or by implementing the JMS semantics in Java.

The main features of a JMS technology that pertain to our research are as follows:

Message Model: A message sent to a JMS provider has three main components. A mandatory header field contains information that the provider needs to deliver and acknowledge messages. An optional set of message *properties* allows the message designer or JMS provider to in essence add new header fields to a message. Properties are defined using the API as name-value pairs. Finally, a message body contains the payload for the message. This can be text, binary, a serializable Java object, a *MapMessage* contain name-value pairs, or a *StreamMessage* whose body contains a stream of Java primitive values.

Publish-subscribe messaging: JMS clients known as publishers can publish messages to the JMS based on topics. Topics are logical addresses that are managed by the JMS provider. JMS subscribers are clients that request messages published on a topic be sent to them. Hence publish-subscribe messaging facilitates N-N messaging in a loosely coupled fashion, in which subscribers and publishers need not know the identity of the message source or destination.

Message Filtering: Subscribers can specify which messages they wish to receive from a topic based on a *message selector*. The JMS provider ensures that only messages that match the criteria in the message selector are delivered to the subscriber. Message selectors are defined by a subscriber on a per topic basis. The selector language is based on a subset of the SQL92 conditional expression syntax. This includes basic conditional and logical expressions, as well as more sophisticated operators such as LIKE.

Importantly, message selectors can only be specified to operate on fields in a JMS header, or defined in a message's properties. This means that message selectors

cannot reference message body values, which can be a severe restriction in some kinds of applications.

3. Improving Message Filtering Performance

3.1 Requirements

We are designing the architecture and technology for a CBM platform based on the JMS with the following application requirements in mind:

Message Sources: Messages will contain text and in general be relatively short. For example, email messages, newsgroup posts and news service (eg CNN) reports and stories will be typical sources of messages. The text of *War and Peace* would be atypical.

Message Format: Due to the diversity of source message formats, the message publisher takes the raw data and converts it in to pre-defined JMS message format.

Message Arrival Rates: Each data source will deliver 100s to potentially 1000s of messages per second. The messages must be processed as quickly as possible, typically within a few seconds of arrival.

These requirements are typical of applications in analytical environments such as finance and intelligence analysis. They have in common the need for rapid filtering of heterogeneous data streams to extract relevant data for subsequent processing.

3.2 Platform Architecture

Figure 1 depicts the basic architecture for the CBM platform. The roles of the four key subsystems are as follows:

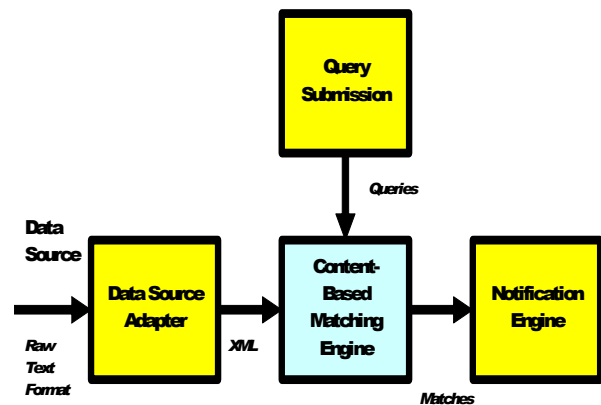


Figure 1 Basic Architecture

Data Source Adapter: The *Data Source Adapter* (DSA) subsystem has several responsibilities. It first inputs messages from the data source in their native format, and transforms them in to a well-defined XML message. In addition, it generates a data signature for the message, which is encapsulated in a Java object. A data signature is a mathematical representation of the textual message, and can be used by the CBM engine to efficiently satisfy queries on the message's content. Finally, it publishes the XML message and associated signature to the CBM engine on a defined topic.

CBM engine: The CBM engine receives messages published by one or more DSA's, with one topic per data source. It also receives queries from the *Query Submission* component. Multiple queries can be registered with a single topic, and hence each message published on a topic must be evaluated against every query. Messages that match one or more queries are sent to the notification engine.

Query Submission: The Query Submission subsystem is responsible for managing the set of queries across the topics handled by the CBM engine. It comprises a base API and a set of user tools for submitting, browsing, deleting and optimizing queries.

Notification Engine: This receives messages that match user queries and delivers them to the respective users. Users may specify a delivery mechanism, such as email, store in a database, and so on. The Notification Engine therefore manages all aspects of notification and message delivery, freeing the CBM subsystem from this responsibility.

The platform is being built using an open source JMS as the core CBM engine. The JMS is being modified internally to exploit new algorithms for rapidly evaluating message selectors (queries), and to incorporate extensions for handling more complex queries on text. The other components utilize the JMS API for publishing and subscribing to messages, and add new capabilities for query management and notification.

The remainder of this paper focuses on the DSA and CBM engine subsystems.

3.2 Data Source Adapter Design

Figure 2 shows the high-level internal architecture of a DSA. It comprises two types of components:

Adapter Manager: Each DSA has a single adapter manager. It supports an external service interface for controlling the adapter from some form of system management infrastructure. This enables the adapter

manager to start up, stop and test the health of individual adapters.

Adapter: Each adapter manager creates and manages one or more adapters. Adapters are responsible for connecting to an actual data source, extracting the data and publishing the formatted XML and data signature to the JMS.

Each DSA runs in a single virtual machine, but is internally multithreaded. One adapter manager can therefore manage one or more adapters, with each adapter handling a unique data source. When instantiating an adapter, the adapter manager sets several configuration parameters. These include the necessary connection information, the JMS topic to publish messages on, and the time delay interval for an adapter between polls of the data source.

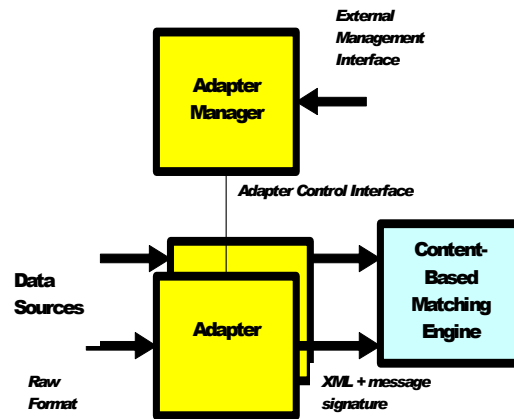


Figure 2 Internal Adapter Architecture

When the adapter retrieves a message from a data source, it performs several actions:

1. It extracts meta-data from the message and embeds this as several properties in the JMS message. Currently the meta-data of interest is items like *author*, *date*, *source*, and so on.
2. It walks through the message text, and removes stop words. Stop words such as *as*, *if*, *it*, are specified in a hash table that is passed to the adapter during construction. This makes it possible to customize a stop word list for a specific data source.
3. Non-stop words are hashed in to a Java `HashTable` object. The hash entry also includes the word's position in the message body. This provides the capability to rapidly answer queries on the message that relate the proximity of two words in the text. This object represents the signature of the message.

4. An XML document is created for the message. It contains tags for each item of meta-data, as well as the full message text. The message text is copied in full, including stop words, into the XML document.
5. It publishes a JMS message to the CBM engine. The message includes the XML document and HashTable representation of the message.

3.3 Message Selector Processing Algorithm

When a subscriber places a subscription to a topic in the JMS, it can optionally specify a message selector. For example, the following message selector would only select messages from CNN written by someone called James.

Author = "James" and source = "CNN"

Internally, when processing a message, the JMS walks through the subscription list and evaluates the message selector associated with each subscription. Each selector is evaluated fully, and if the selector is true, then the message will be delivered to that subscriber. This scheme is depicted in Figure 3.

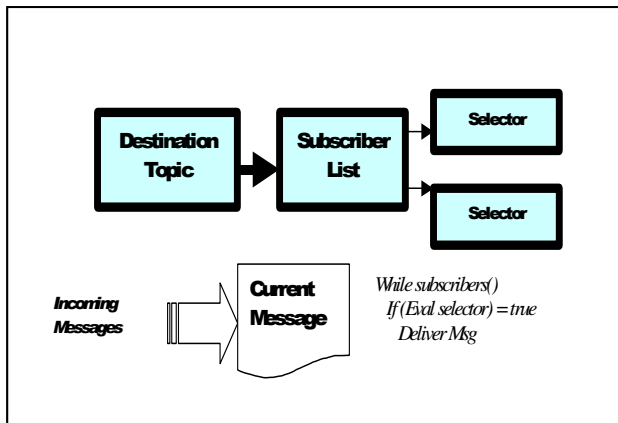


Figure 3 Processing Message Selectors in the JMS

As each selector is evaluated in isolation, the time required to process a message is proportional to the number of selectors, as well as the cost of evaluating each individual selector. This situation is non-scalable. As the number of selectors on a topic becomes large, message throughput must degrade significantly.

In order to improve message selector performance, the CBM engine modifies the internal evaluation algorithm of the JMS. Essentially, rather than having independent selectors which each have to be individually evaluated to completion, the algorithm exploits shared elements in selectors. This completely eliminates the need for

duplicate evaluations for the same message, and consequently provides significantly better performance.

When a new selector is added to a topic, for example:

Author = "James" and source = "CNN"

it is broken down in to a set of individual elements. These are termed *cacheable units*, and in this example they would be as follows:

Unit 1: Author = "James"

Unit 2: source = "CNN"

Next, the algorithm searches to see if any of these new cacheable units are shared with other selectors. If so, it replaces the cacheable unit with the common one, producing a consolidated message selector. This is illustrated in Figure 4.

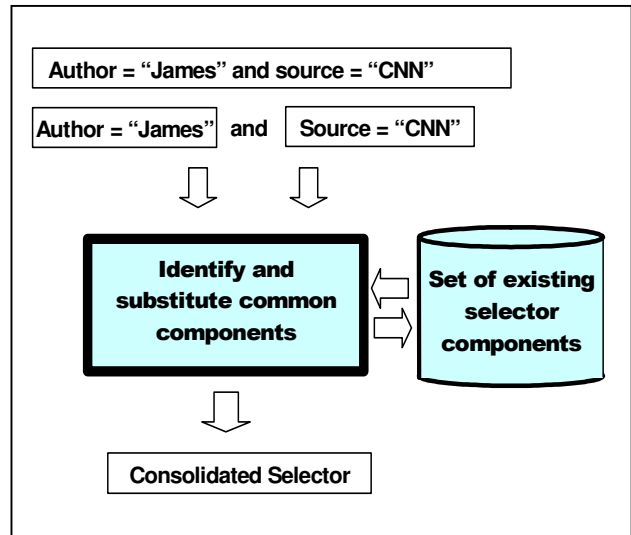


Figure 4 Exploiting common elements in message selectors

Now, during message processing, a publisher sends a message to the topic and the server tests it against all the registered selectors. When a selector is evaluated against the message properties, the result of each cacheable unit is stored in a cache data structure, which maps a cacheable unit to a Boolean result.

As each message is tested against all the other registered selectors, the cache is accessed in order to locate any shared selector elements that have already been evaluated. When a cacheable unit is associated with a result for this message, its result is simply used in evaluating the current selector. Hence, the commonality between selectors is exploited to reduce the number of

queries that must be satisfied, and produce greater message throughput.

4. Performance Tests

4.1 Test Case Description

A test suite was developed to measure and quantify the performance differences between the base JMS and improved CBM engine. The test suite's aim is to show the improvement in performance possible due to the new selector evaluation algorithm.

The test suite sets up an environment in which publishers and subscribers can run with a given set of selectors against the modified content-based matching engine versus the original JMS implementation. The JMS implementation used is the JBoss JMS (<http://www.jboss.org/developers/projects/jboss/jbossmq.jsp>). The CBM engine modifies the JBoss JMS with new message selector algorithms.

The test suite was designed to measure the maximum sustainable throughput as described in [8] for each version of the JMS implementation. More specifically, the receive rate was emphasized in order to compare and benchmark the two JMS implementations.

The publishing task is accomplished through one or more publisher instances that all publish at a given rate. This makes it possible to simulate high messages loads being sent to the CBM engine.

The publishers all post the same message to the same JMS topic, to which all the subscribers listen. The purpose of posting the same message repeatedly is to ensure known data, and hence response, and to stress the message selector algorithm throughput. The test message for the test suite is modeled after an Internet newsgroup posting, with JMS properties for the *sender*, *date*, *subject*, and *message body*. The actual JMS message payload is left empty since it is not used by the message selector algorithms. Figure 5 is an example message that is posted:

```
From = "Someone (Someone@yahoo.com)"
Date = "2002-12-28 21:04:22 PST"
Subject = "I have a question..."
Body = "I covered my daffodils, garlic, and rhubarb
last night in hopes of saving them from freezing.
Should one uncover them during the day if the temp
is above freezing, and re-cover them in the late
afternoon? Or can I just leave them covered for a
few days till the nights get warmer? (may be a moot
point after I look at them...) "
```

Figure 5 Example Test Message Format

As described previously, the selector evaluation algorithm takes advantage of overlapping queries to increase the performance of the matching engine. Consequently, the test scenario is designed to investigate the best achievable performance in optimal conditions for this algorithm. To this end, each subscriber registers the same selector on the same topic. This should present an ideal test case for the CBM engine.

In addition, the test case explores the impact of the cost of evaluating individual selectors. At one extreme, very simple queries that are inexpensive to evaluate were tested, and at the other extreme more complex queries that are expensive to evaluate were created.

At the least expensive end of the spectrum is a query that runs against a small text field and performs an exact match. An example simple selector is:

```
From = "Someone (Someone@yahoo.com)"
```

This simple selector is labeled *selector expense level 1* in the results that follow.

A slightly more expensive query is one that still operates on a small text field, but that uses a more expensive operator. In this case, the more expensive operator is the "LIKE" operator that supports wildcards for finding substrings within text. An example selector for this case is as follows:

```
Subject LIKE "%question%"
```

This type of selector is labeled *selector expense level 2* in the results.

A third level of selector expense involves searching within a much larger text field while using the LIKE operator, but where the match is near the beginning of the text. The length of the text greatly influences the speed at

which LIKE can execute, thus making the field larger will cause the operation to be much more expensive. An example selector for this case, and *labeled selector expense level 3* in the results is:

```
Body LIKE %garlic%
```

The most expensive operator for the test suite is one that searches a large block of text using wildcards to locate a certain word or word fragment. This operation is very expensive because the LIKE must search the entire block of text to determine if a match is made. Since in our test data, the match is near the end of the text block, it will be expensive and time consuming to evaluate. An example selector for this case, which is labeled *selector expense level 4* in the results is:

```
Body LIKE %warmer%
```

The performance tests were run on hardware with the following configuration:

Publisher node: Pentium III 1.3 GHz with 768MB of RAM running Windows XP and Java 1.3.1

Subscriber node: Pentium III 1.3GHz with 768MB of RAM running Windows XP and Java 1.3.1

CBM Server: Quad 2GHz Pentiums running Windows 2000 Server and Java 1.3.1

To alleviate clock synchronization issues in the measurement of performance, all publishers and all subscribers were run on the same machine. Additionally, the publishers and subscribers were run within the same Java Virtual Machine, but on different threads. For all test cases, there were 10 publishers publishing at the maximum sustainable throughput rate, which is dependent upon the JMS throughput. The number of subscribers is listed for each test case. This dictates the amount of overlap among message selectors, which is an important element in the faster message selector processing algorithm.

Since the focus of the CBM engine is to handle large amounts of streaming data, it is assumed that speed should be favored over reliability when it comes to message delivery. Consequently, all messages were published with a *delivery mode* of NON_PERSISTENT, which indicates that the JMS provider does not need to persistently store the message to ensure delivery in the case of server failure. Moreover, this also eliminates any decreases in performance due to disk access and allows the test to concentrate solely on the message selector processing algorithm.

4.2 Results

Figure 6 through Figure 9 show the performance results for varying numbers of subscribers. For these cases, 100% of the messages sent by the publishers match the selector for each subscriber, thus all messages are sent out to subscribers.

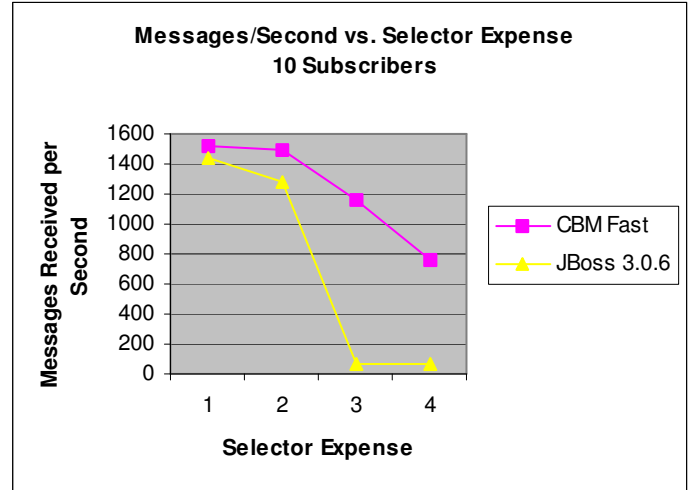


Figure 6 Throughput with 10 Subscribers

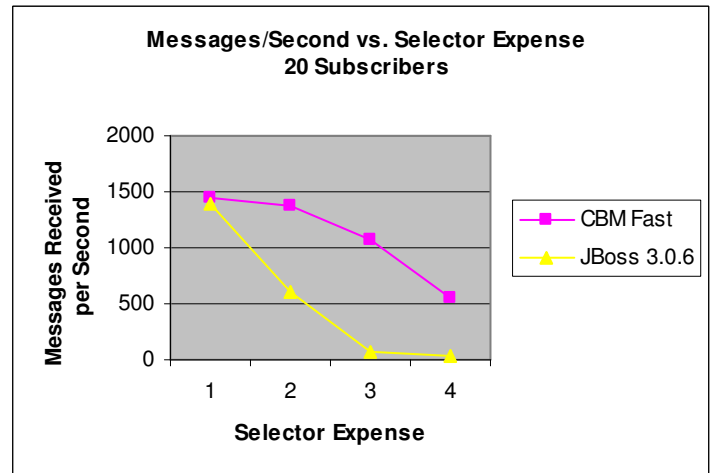


Figure 7 Throughput with 20 Subscribers

For the simplest selector, there is little difference in performance between the JBoss JMS and the modified CBM engine. However, as selectors become progressively more complex to evaluate, the performance of the JMS rapidly degrades. As an example, with 100 subscribers, the JBoss JMS throughput decreases from 1082 messages per second with the simple selector, to 18.6 messages per second for the most complex selector. This represents a drop in performance of approximately 98%. In comparison, the CBM engine peaks at 1202

messages per second for the simple selector, and degrades to 721 messages per second for the most complex. This represents only a 40% drop in performance.

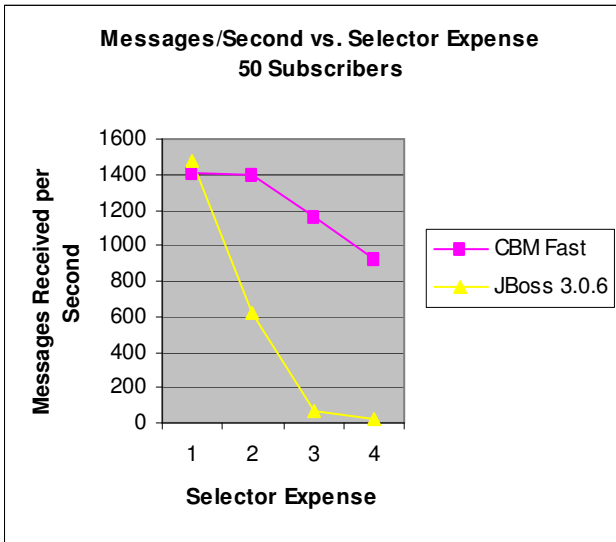


Figure 8 Throughput with 50 Subscribers

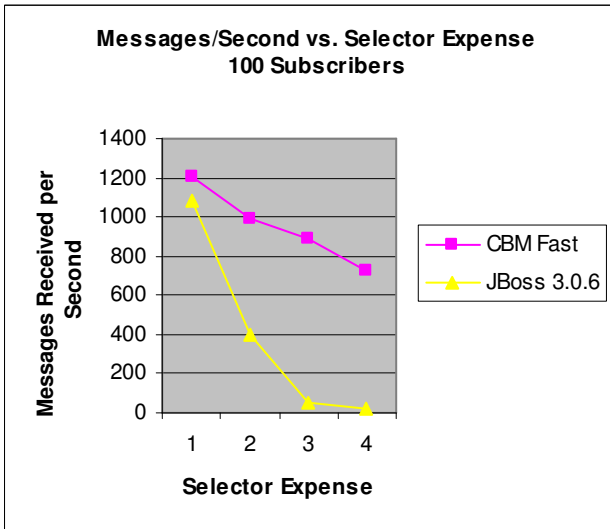


Figure 9 Throughput with 100 Subscribers

These test results therefore clearly show the inherent limitations in the original selector processing algorithm. As the expense of the selector increases, the JMS implementation is unable to sustain a useable message throughput level.

Alternatively, the improved selector processing algorithm demonstrates a basically linear regression in performance, which creates a stable and reliable content-based messaging system.

Figure 10 and Figure 11 show a more typical JMS usage in which only 10% of the subscribers have selectors that will match the message being published. This represents a test case that is less biased to exploiting the performance improvements that have been introduced in to the CBM engine. In addition, the number of subscribers ranges from 200 to 400 on the topic under test.

The results importantly show the same trend as the previous test case. The JBoss JMS implementation exhibits a rapid decrease in throughput as the selector expense increases, while the CBM implementation handles increased selector expense in linear fashion.

Interestingly, for the simple selector, the JMS slightly outperforms the CBM engine in both tests. This is illustrated with 400 subscribers, where the JMS throughput is 662 compared to 631 messages per second for the CBM engine. The cause for this is the fact that the CBM algorithm introduces a small overhead to selector processing. However, for the most expensive selector test, the JMS throughput decreases to 11 messages per second, whereas the CBM engine throughput drops to 393, or 62% of peak throughput compared to less than 2% for the JMS.

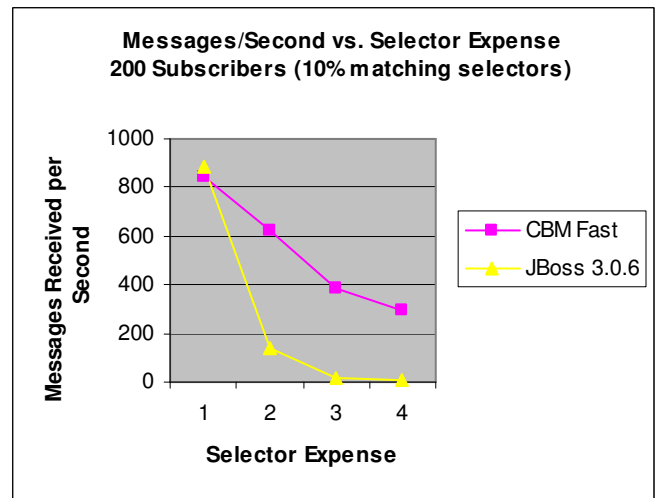


Figure 10 Throughput with 200 Subscribers

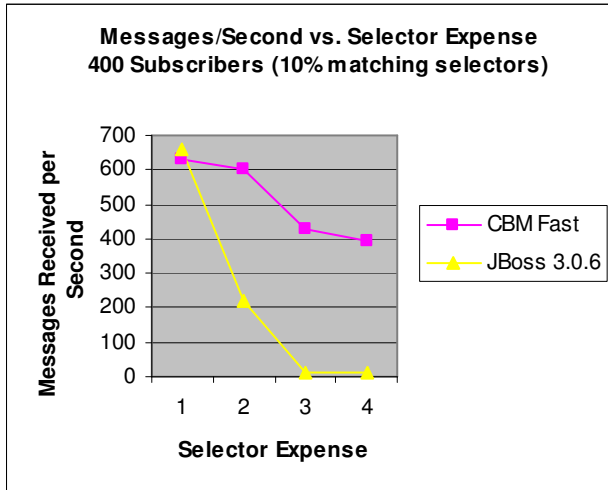


Figure 11 Throughput with 400 Subscribers

5. Conclusions and future work

By introducing new algorithms in to a JMS implementation that exploit commonality in message selectors, this project has clearly demonstrated the performance and scalability gains that can be achieved. This is an extremely positive result that shows how standard JMS implementations can be improved to make them faster for content-based messaging.

Our current work is introducing more new algorithms in to the JMS that exploit the message signature that is produced by the CBM platform adapters. The use of the signatures should improve the performance of every basic selector evaluation. This promises to have a major impact on the message throughput that is achievable. It is however not possible to achieve this within the current JMS specifications, as message properties are restricted to simple types, and cannot be Java objects. Hence minor extensions to the JMS will be required.

6. References

- [1] Babcock, Brian; Babu, Shivnath; Datar, Mayur; Motwani, Rajeev; Widom, Jennifer. Models and Issues in Data Stream Systems, Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS 2002)
- [2] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System In Proc. of the 2003 Conference on Innovative Data Systems Research (CIDR), January 2003
- [3] Bill Segall and David Arnold, "Elvin has left the building: A publish/subscribe notification service with quenching," *Proceedings AUUG Technical Conference (AUUG'97)*, pp. 243-255 (September 1997).

- [4] Marcos K Aguilera, Robert E Strom, Daniel C Sturman, Mark Astley, and Tuschar D Chandra, "Matching Events in a Content-based Subscription System," *Principles of Distributed Computing*, (1999).
- [5] Marcel Ruff, *White Paper xmlBlaster: Message Oriented Middleware (MOM)*, <http://www.xmlblaster.org/xmlBlaster/doc/whitepaper/whitepaper.html> 2000.
- [6] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service". *ACM Transactions on Computer Systems*, 19(3):332-383, Aug 2001.
- [7] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service". *19th ACM Symposium on Principles of Distributed Computing (PODC2000)*, Portland OR. July, 2000.
- [8] P. Tran, P. Greenfield, I. Gorton, Behavior and Performance of Message-Oriented Middleware Systems, *Proceedings, 22nd Int'l Conf on Dist. Computing Systems Workshops*, Vienna 2-5 Jul 2002. Pages 645-650, IEEE