

Integration of a Text Search Engine with a Java Messaging Service

Justin Almquist¹, Ian Gorton², Jereme Haack¹

¹Information Sciences and Engineering, Pacific Northwest National Lab, Richland, WA 99352, USA

²Empirical Software Engineering Group, National ICT Australia, Australian Technology Park, Eveleigh, NSW 1430, Australia

Abstract. Large-scale information processing applications must rapidly search through high volume streams of structured and unstructured textual data to locate useful information. Content-based messaging systems (CBMSs) provide a powerful technology platform for building such stream handling systems. CBMSs make it possible to efficiently execute queries on messages in streams to extract those that contain content of interest. In this paper, we describe efforts to augment an experimental CBMS with the ability to perform efficient free-text search operations. The design of the CBMS platform, based upon a Java Messaging Service, is described, and an empirical evaluation is presented to demonstrate the performance implications of a range of queries varying in complexity.

1. Introduction

Efficiently finding useful data in the ever-growing sources of digital information is a highly challenging research problem. Many applications must search for messages or packets of information located, for example, in network protocol traffic or email messages. Such applications are typically known as data stream processing applications.

Processing continuous data streams [1] poses some unique problems. A streams processing environment must assume that the messages it receives are transient, and cannot all 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.

Content-based messaging systems (CBMSs) have proven practical technology platforms for building data stream processing applications. A CBMS employs some mechanism for querying the content of an individual message and extracting messages from the stream that satisfy one or more query. 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, but in general most utilize a relatively simple query language.

In addition, 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.

In [9], we describe our efforts to improve the performance of an open source JMS implementation to provide scalable performance when multiple simultaneous queries must be evaluated against each message in a stream. Algorithms similar to those described in [4, 6, 10] are empirically demonstrated to provide greater than order of magnitude performance improvement and significantly improved scalability when compared to the original JMS implementation.

In practice however, the query language supported by the JMS has proven too simple to be of great utility in application environments. Therefore, this paper describes the extension of our JMS-based CBMS to integrate with an off-the-shelf high performance text search engine. We describe the approach taken to extending the JMS query capability and utilizing the text processing engine. An empirical evaluation of the performance of the resulting platform clearly shows the feasibility of the approach. Further performance analysis factors out the influence of the underlying JMS implementation, and the results display potential for achieving even greater performance from a CBMS platform built from a combination of a JMS and text search engine.

2. A JMS-based Content-Based Message System

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 publish-subscribe 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.

JMS 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.

[9] describes how we extended the open source JBoss JMS implementation to form a CBMS that can efficiently process multiple streams of data. In summary, the extensions are briefly described below.

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

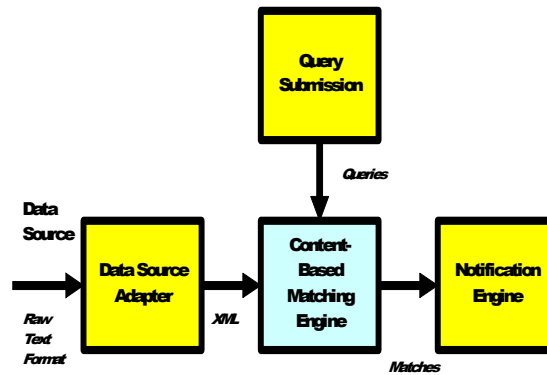


Fig. 1. Overview of CBMS Architecture

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 CBMS 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. Additionally, the Notification Engine is responsible for persisting matched messages. This allows relations between users, queries, and messages to be established such that end users can determine which queries caused certain messages to be matched and saved.

The core CBM engine comprises the modified JBoss JMS. The JMS was modified internally to exploit new algorithms for rapidly evaluating message selectors (queries). The algorithms basically break down each message selector (query) in to a

set of individual elements, which are termed *cacheable units*. When a selector is evaluated against the message content, 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. Figure 2 illustrates the improved performance that the CBMS platform achieves. These are more results are fully explained in [9].

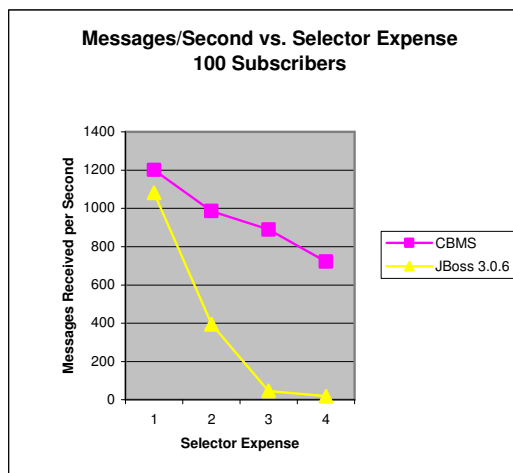


Fig. 2. CBMS message throughput with 100 Subscribers and increasing complexity in message selector evaluation.

3. Extending the JMS Query Language

3.1 The Need for Powerful Text Search

In discussions with potential users of the CBMS platform, it became clear that existing commercial technologies, while slower, expensive and heavyweight, offered much greater flexibility and power in free-text search queries. We therefore decided to investigate how the CBMS platform could incorporate a richer query language in order to increase the relevance of messages delivered to users. At the same time, it remained imperative that the query language did not degrade significantly the performance and scalability gains already achieved. After investigating the

capabilities of existing technologies in the user's application domain, our goal became to extend the current JMS query language with greater free-text searching capabilities.

High performance text searching engines are highly specialized technologies, and hence writing our own was not a sensible option. An investigation of available technologies led us to decide to integrate the open source Lucene¹ full-featured text search engine into the CBMS. Lucene supports a simple API, is extensible, has small heap and index size requirements, and claims high performance and scalability. It also has the following extensive text query features²:

- *Terms* - A Single Term is a single word such as "test" or "hello". A Phrase is a group of words surrounded by double quotes such as "test hello".
- *Wildcard Searches* – Using the '?' will match a single character, such as "te?t". Using the '*' will match multiple characters, such as "test*".
- *Fuzzy Searches* - The fuzzy search features uses the tilde, "~" symbol at the end of a Single word Term. For example, a query with "roam~" would return terms like foam and roams.
- *Proximity Searches* - Lucene supports finding words that are within a specific distance away from each other.
- *Boolean Operators* – The traditional Boolean operators of "AND", "OR", and "NOT" are supported, as well as the "+" (required) and "-" (prohibit) operators.

3.2 Messaging with Lucene

In order to investigate the implications of integrating Lucene into the CBMS, we decided to initially extend the query language supported by the JMS standard. The JMS specification's keywords were consequently extended with a new keyword, `INCLUDES`. Extending the specification is attractive, as it allows the CBMS to still meet the JMS specification, but at the same time extends the JMS functionality for applications that require advanced query behavior.

To simplify the implementation, only queries using the `INCLUDES` keyword would result in a call to the Lucene API. Hence the internals of the query processing implementation of the CBMS were left unchanged, and the overheads of utilizing Lucene were only incurred when necessary.

Integrating Lucene also required design decisions to be made regarding Lucene's usage. The single largest impact on performance when using a text search engine is the indexing of the source data. In normal modes of application, indexing is an expensive process that requires analyzing many documents and writing the results of the analysis to a disk-based index. Lucene incorporates a high performance indexing mechanism; however any indexing scheme that would have to write files to disk would be prohibitively slow and non-scalable.

Thus, it was determined to use memory based indexes instead of disk-based. This exploited the same optimized indexing and avoided the overhead of disk I/O. This

¹ (<http://jakarta.apache.org/lucene/docs/index.html>)

² (<http://jakarta.apache.org/lucene/docs/queryparsersyntax.html>)

design is especially appropriate since an index is created for every message, and can be discarded when the message has been fully processed.

Another key design decision in exploiting the Lucene engine is the ability to index a message once and run multiple sub-queries against the index. A complex query that is typical in the CBMS application environment would comprise many grouped sub-queries such as:

```
(body INCLUDES "flour") AND (body INCLUDES "sugar") AND  
(body INCLUDES "peanuts")
```

Using Lucene, in our solution the message is indexed once, which is expensive, and then each subsequent sub-query searches the index, which incurs a minimal cost. This should lead to complex queries having approximately the same performance as simple queries, which is an extremely desirable property for the CBMS.

One area that was not explored was that of the analyzer performance. Indexing and subsequent searching in Lucene utilizes a class that implements a *StandardAnalyzer* interface. Since analyzers are involved in all expensive operations (indexing and searching) it is imperative that they are optimized. However, lack of time prevented the further testing of different analyzers to measure performance differences.

4 Performance Analysis of Text Queries

4.1 Test Case Description

In order to verify that the use of text search engine does not impact performance and scalability of the CBMS, a series of performance tests were carried out. The same test suite used to verify the improved JMS query engine [9] was used to measure and quantify the performance impact of using the Lucene search engine. Our goal was to demonstrate that the Lucene based text queries execute at least as fast as the original JBoss implementation while providing expanded querying capabilities. Hence we designed and executed 2 sets of equivalent tests to compare the performance of the JBoss JMS executing `LIKE` queries with the CBMS features that exploited Lucene, namely the `INCLUDE` queries.

The test suite creates multiple instances of publishers to simulate increasing message loads. The same message is posted to the JMS topic that all subscribers listen, thus creating an environment with known input data. For each message, subscribers specify message selectors of varying complexity. For the JBoss JMS tests, the selectors utilize the keyword `LIKE` along with wildcard indicators ("`%`"). The Lucene engine in the CBMS is tested by selectors utilizing the `INCLUDES` keyword, as well as the Lucene wildcard characters ("`*`").

To explore the impact of the cost of evaluating individual selectors, the series of tests uses selectors of varying complexity. At one extreme, very simple selectors that

are inexpensive to evaluate were tested. Subsequent tests used progressively more complex selectors to examine the effects of their evaluation on the JBoss JMS and CBMS with Lucene. These are explained below.

For the JBoss JMS tests, the least expensive is a query that runs against a small text field, such as the subject field of an email. An example simple selector is:

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

This simple selector is labeled *selector expense level 1* in the presentation of the performance results.

A slightly more expensive query is one that operates against a larger text field, such as the body of the message. An example selector for this case is as follows:

```
Body 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, but where the match is near the end of the text. The length of the text greatly influences the speed at which wildcards can execute, thus making the field larger will cause the operation to be much more expensive to execute. An example selector for this case, and labeled *selector expense level 3* in the results is:

```
Body LIKE "%garlic%"
```

The fourth most expensive operator for the test suite is one that searches a large block of text using two sub-queries with wildcards to locate certain words. This operation is more expensive because both wildcard expressions must search the entire block of text to determine if a match is made. An example selector for this case, which is labeled *selector expense level 4* in the results is:

```
Body LIKE "%garlic%" AND  
Body LIKE "%question%"
```

The fifth selector level increases the number of sub-queries to three, which will require all three words to be found in the large block of text. An example for *selector level 5* is:

```
Body LIKE "%garlic%" AND  
Body LIKE "%question%" AND  
Body LIKE "%afternoon%"
```

The most expensive selector level is a query that finds many words in a given text block. This type of complex query is expected to be typical in the environment the CBM is being constructed for. An example for *Selector level 6* is:

```
Body LIKE "%garlic%" AND  
Body LIKE "%question%" AND
```

```
Body LIKE "%afternoon%" AND  
Body LIKE "%freezing%"
```

Equivalent queries were also constructed for the CBMS with Lucene, utilizing the `INCLUDE` keyword instead of `LIKE`. Hence, when the two sets of queries are executed on identical data, the same set of results is produced.

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

- **Publisher/Subscriber node:** Pentium III 1.3 GHz with 768MB of RAM running Windows XP and Java 1.3.1
- **CBMS/JBoss Server:** Quad 2GHz Pentiums running Windows 2000 Server and Java 1.3.1

To alleviate clock synchronization issues in the performance measurement, 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. The CPU utilization was low however, and did not influence tests results. For all test cases, there were 10 publishers publishing at the maximum sustainable throughput rate [8], which ensures messages do not build up in the JMS queues and degrade performance. The number of subscribers is listed for each test case.

Since the focus of the CBMS 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

Figures 3 and 4 show the performance results for increasing 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.

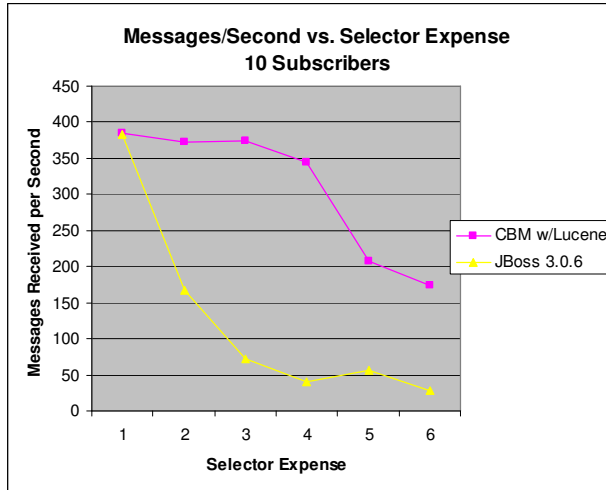


Fig. 3. Throughput with 10 subscribers

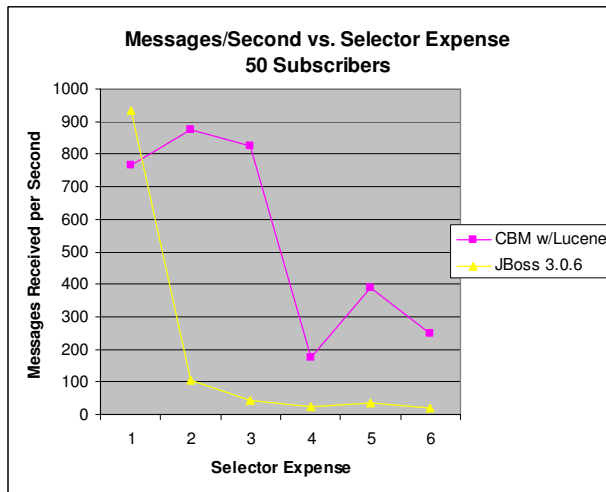


Fig. 4. Throughput with 50 subscribers

In Figure 3, for the least inexpensive selector, the CBMS and JBoss have little difference in respective throughput. However, as selectors increase in complexity, the performance of JBoss degrades rapidly from 384 messages per second with the simplest selector to 28 messages per second for the most complex selector. Thus, JBoss' performance drops by approximately 93%. Alternatively, the CBMS with the Lucene engine peaks at 385.6 messages per second for the simplest selector, and slows to 174.48 messages per second for the most complex, which represents only a 55% drop in performance.

As the number of subscribers increases (Figure 4), the CBMS with Lucene clearly outperforms the original JBoss. Moreover, the original JBoss engine is completely unable to handle increasingly complex selectors and degrades 98% while the CBMS with Lucene drops in performance by only 67%.

These test results clearly demonstrate that the introduction of the Lucene search engine as the selector matching engine does not negatively impact the performance of the original JBoss implementation. Rather, Lucene performs better than the original JBoss implementation in terms of throughput and scalability. Thus, the original goal of increasing query expressiveness in order to improve message matching has been achieved with no penalties in terms of performance or scalability.

4.3 Analysis

The original JBoss wildcard engine is implemented by a regular expression package. Essentially, the selector is converted to a regular expression and then passed on to the regular expression library to be handled. However, regular expressions are quite expensive to calculate and this limitation clearly shows as the complexity of the selector increases.

Conversely, the CBMS with Lucene performance does not degrade so severely as the selector complexity increases. This is due to the fact that the matching engine uses Lucene to create an index for each message. Then, each sub-query against that message is run against the original index. The cost of searching against an index is much less than the cost of evaluating a regular expression. Thus, the performance of CBMS with Lucene is dominated by the cost of creating the index for each message.

Therefore, the benefit of utilizing Lucene for complex selectors combined with the earlier selector caching algorithms make for a fast, scalable platform for content-based matching.

5. Analyzing Lucene Performance

Another test suite was developed to measure the differences between Lucene and the CBMS with Lucene. The aim was to assess the maximum performance that might be achievable with Lucene, and quantify the overheads introduced by the underlying JBoss JMS platform.

To this end, one test environment was setup in which text messages are handled by the CBMS utilizing JMS publishers and subscribers (as described earlier). A second test environment simply delivered messages to a Java program that called the Lucene engine. The latter hence aimed to illustrate the performance potential of Lucene without any influence from the JMS.

The same input messages were used for both environments. Likewise, the same selectors were setup for the content based matching algorithms, with varying selector complexities.

Table 1 shows the selectors used.

Deleted: ¶
¶
Table 1

Table 1 Selector Complexities

Selector 1	subject INCLUDES 'atheism'
Selector 2	body INCLUDES 'might'
Selector 3	body INCLUDES 'angry'
	body INCLUDES 'might' AND
Selector 4	body INCLUDES 'angry'
	body INCLUDES 'might' AND
	body INCLUDES 'angry' AND
Selector 5	body INCLUDES 'important'
	body INCLUDES 'strange' AND
	body INCLUDES 'important' AND
	body INCLUDES 'might' AND
Selector 6	body INCLUDES 'angry'

Both tests were executed on the same hardware platform³, and all test components were executed on the same machine. Figure 5 shows the results obtained.

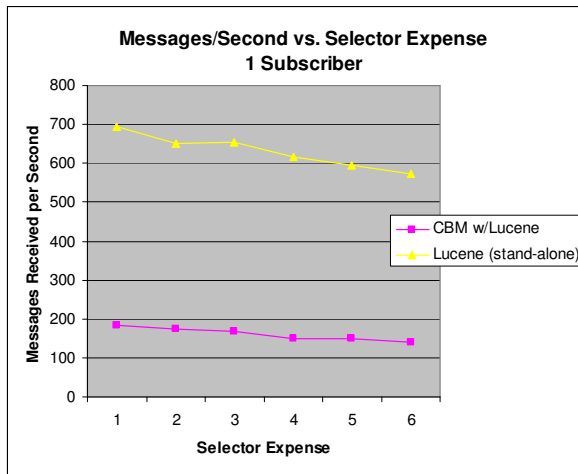


Fig. 5. Lucene vs. CBMS with Lucene Results

The results starkly illustrate the JMS implementation overhead. For this set of test data and queries, the Lucene search engine is able to process 694.52 messages per second at the lowest selector complexity, while the CBMS with Lucene is only able to sustain 184.52 messages per second. This represents a 73% performance degradation.

At the time of writing, we have not had time to investigate thoroughly the cause of the overheads in the JBoss JMS. However, in associated tests with other JMS

³ Pentium IV 3 GHz with 1GB of RAM running Windows XP and Java 1.3.1

platforms, greater JMS message throughput has been observed. Hence we have a good degree of confidence that a faster JMS implementation is possible, and that such an implementation may be able to fully exploit the performance potential of the Lucene engine.

6. Further Work and Conclusions

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.

In this paper, we have demonstrated the feasibility of further integrating a full-featured text search engine, Lucene, with a JMS implementation. Empirical testing shows that Lucene can provide excellent, scalable performance. This gives us confidence that we can go on to implement a CBMS based upon the messaging features of a JMS combined with extensions of the JMS query capability to exploit powerful full-text searching.

We are continuing to explore content-based matching with JMS infrastructures. To this end, we are performing a set of experiments that attempt to factor out the transport cost of a particular JMS. This will enable us to directly compare the costs of our content-based text matching engine with those of several JMS implementations. We also intend to scale our CBMS platform to run across a cluster of machines to ensure a large number of subscriptions can be handled.

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] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In Eighteenth ACM Symposium on Principles of Distributed Computing (PODC '99), pages 53--61, Atlanta, Georgia, May 4--6 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 (PODC 2000)*, 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
- [9] I.Gorton, Justin Almquist, Nick Cramer, Jereme Haack, Mark Hoza, An Efficient, Scalable Content-Based Messaging System, in Procs The 7th IEEE International Enterprise Distributed Object Computing Conference, (EDOC 2003), pages 278-285, Brisbane Sept 2003
- [10] Francoise Fabret, H.-Arno Jacobsen, Francois Llibat, Joao Pereira, Kenneth Ross, Dennis Shasha, Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. ACM SIGMOD 2001 Conference, Santa Barbara, pages 115-126, CA. May, 2001.

Acknowledgements

This work has been funded by PNNL's Energy Sciences & Technology Directorate Lab Directed Research & Development Program.

National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council