Towards Improved Data Dissemination of Publish-Subscribe Systems

Ramith Jayasinghe, Dinesh Gamage, Srinath Perera
Lanka Software Foundation
Colombo 07, Sri Lanka
{ramithj, dineshg, hemapani}@opensource.lk

Abstract—with the proliferation of internet technologies, publish/subscribe systems have gained wide usage as a middleware. However for this model, catering large number of publishers and subscribers while retaining acceptable performance is still a challenge. Therefore, this paper presents two parallelization strategies to improve message delivery of such systems. Furthermore, we discuss other techniques which can be adopted to increase the performance of the middleware. Finally, we conclude with an empirical study, which establishes the comparative merit of those two parallelization strategies in contrast to serial implementations.

Keywords- publish-subscribe system; web services; ws-eventing; ws-notifications; parallel processing

I. INTRODUCTION

In recent years, distributed systems have increasingly adopted message based models, thus switching from RPC based, synchronous request-response interactions to more generic asynchronous and one way messaging interactions. Such messaging interactions have enabled many large scale systems, where the need for loose coupling is a fact, rather than a choice.

Among such messaging models, publish-subscribe systems [1] provide loose coupling between participants, allowing them to co-exist with minimum knowledge about others. A publish subscribe system is typically implemented as one or more message brokers, which matches messages sent from publishers with subscriptions and delivers them to interested parties. On the other hand, service oriented architecture (SOA) has gained wide acceptance as an integration and implementation technology for distributed systems. Consequently, two competing specifications namely, WS-Eventing [4] and WS-Notifications [5] emerged to define publish-subscribe systems specifically within SOA.

To be used to integrate large scale distributed systems publish-subscribe systems themselves should scale. Therefore, building scalable publish-subscribe systems has been a useful problem for almost a decade. However, still building such scalable systems has continued to be a challenging task.

This paper focuses on message delivery, one of the bottleneck prone part of publish-subscribe system design, and present few potential approaches and detailed analysis of their comparative merit. The presented study is motivated by WS-Messenger [2][3], a message broker that supports publish-subscribe paradigms for SOA by implementing both aforementioned specifications. It has been implemented as a part of the LEAD [7] project—a large scale E-Science project that provides U.S. wide weather processing. Using the WS-Messenger within LEAD has posed significant scalability requirements, and we draw experience for this study from the integration of WS-Messenger and LEAD.

The rest of the paper is organized as follows. The next section discusses related work in the field, and the following section III discusses the high level architecture of WS-Messenger. Section IV presents our observations that describe few commons problems found in a typical message broker and the following Section 5 discusses potential solutions for them. The section VI presents an empirical study that tries to measures the merits and effectiveness of each approach. Finally, the section VII concludes the paper.

II. RELATED WORK

Improving performance while maintaining the scalability in publish-subscribe systems has been addressed by many, previously. However, most of this work is focused on how XML data streams can be filtered efficiently. These attempts to filter XML messages broadly fall in to three categories; where first approach was to develop algorithms to match subscriptions in an efficient way. One of the notable contributions taking this approach is Yfilter [11]. Yfilter is improved version of Xfilter [16] which builds a single nondeterministic finite automaton (NFA) to evaluate Xpath expression submitted by users; while Xfilter builds finite state machine (FSM) for each Xpath expression submitted. It should be noted that WS-Messenger uses Yfilter for XML filtering.

Secondly, Performance of the XML filtering can be further improved by incorporating special Hardware solutions based on FPGAs as suggested in [14]. By implementing Xpath queries on FPGAs, this solution overcomes bottleneck imposed by the limited throughput (data transfer rate) between the CPU and main memory which is inherent to any software based solution hence improving the performance drastically.

Thirdly, motivated by current trends of hardware manufacturers to include more processing cores in a single chip, [13] presents three parallel algorithms to reduce matching time or increase throughput. The paper also discusses how these algorithms are implemented using traditional locks as well as software transactional memory.
But, with WS-Messenger we experienced that filtering data as well as delivering filtered messages amounts to a significant load. Additionally, inadequate delivery mechanisms results in an increased round trip time of messages as well as a reduced throughput. If not addressed carefully, this will become a major bottleneck for the system’s performance. But this issue is yet to be addressed by researchers.

III. HIGH-LEVEL ARCHITECTURE OF WS-MESSENGER

WS-Messenger is a part of the open grid computing environments project. Currently, it is used as the messaging subsystem of Linked Environments for Atmospheric Discovery (LEAD) project, which is the ancestor of OGCE project.

WS-Messenger supports both topic and Xpath [8] based subscriptions. Additionally, the current version can be deployed either as a standalone distribution or as a servlet within a servlet container such as Apache Tomcat. Earlier system used XSUL [6]–a web services framework developed by Indiana University. However, to support industry standards, system was ported to Apache Axis2 web services stack.

In order to distribute load, WS-Messenger is divided into two separate components, where the first component receives messages from publishers and matches them against topic and Xpath expressions, while the second component sends out matched messages to consumers in a serial manner.

IV. OUR OBSERVATIONS

Our analysis of the system revealed a number of issues in terms of performance. However, we believe these issues are not only specific to this system but also to other message brokers which are designed to work in heterogeneous environments. Let us we discuss some of the major challenges associated with such systems in the following paragraph.

A. Limitations associated with the existing systems architecture

WS-messenger receives a massive number of messages from publishers. Once filtered, these messages get accumulated in the memory until they are delivered to interested consumers. If the message dissemination rate is less than the message reception rate, a memory growth is guaranteed. As the memory usage increases, the system tends to lag in performance, hence reducing the throughput. Additionally, this increases the round trip time of messages. Main reason for this bottleneck is the serial nature of message dissemination as shown in Figure 1.

B. Limitations associated with the network

Message publishers and consumers who interact with WS-Messenger are distributed over a heterogeneous environment. Therefore the nature of connectivity between consumers/publishers is essentially unpredictable and beyond the control of the system. Hence, middleware should be able to prevent these connectivity problems from affecting its overall performance.

For example suppose, there exist a set of consumers, where the connectivity between system and consumers are slow compared to others. This will lead to a reduced throughput in the message delivering component, since the time taken to deliver a message to a slow consumer is larger compared to others. Consequently, this will increase the round trip time of messages for faster consumers as well.

Furthermore, some error situations will also lead to a reduced performance of the system. Suppose connectivity between a consumer and WS-Messenger timeouts every time messenger tries to deliver a message. Then the thread used to send out the message will be blocked until the end of the timeout period. Trying to resend messages blindly to such consumers will worsen this adverse effect. Unfortunately, this behavior is inherent in the previous architecture of this publish/subscribe system.

C. Blocking I/O

Originally, the message receiving component of the middleware was implemented on top of the blocking I/O mechanisms provided by Java API. This acts as a bottleneck when WS-Messenger encounters a large number of concurrent connections.

V. OPTIMIZATIONS

A. Consumer blacklisting scheme

In order to minimize the overhead incurred by message delivery failures, we propose a smart blacklisting scheme as described below.

At the beginning, identified errors related to message delivery are divided into two categories where one being errors eligible for blacklisting and other as being not eligible for that. This can be configurable. Upon a message delivery failure, if the error is configured as eligible, then the system associates a counter with the respective consumer, but if a counter has been previously associated, then the system simply increment that. After the counter exceeds a pre-configured threshold during subsequent delivery failures, the consumer is treated as blacklisted. System will not try to deliver subsequent messages to blacklisted consumers for a certain period of time.

Meanwhile a background thread will periodically decrease the counters of blacklisted consumers, effectively clearing the blacklist flag. Note that counter is not removed by the background thread; rather it is removed only if a subsequent message (after decreasing the counter) delivery becomes a success. To make this scheme successful user must correctly configure types of delivery errors which will affect the system’s performance negatively.

Figure 1 Serial message delivery
B. Non-blocking I/O and concurrent message filtering

In order to reduce overhead imposed by blocking I/O, we configured an NIO transport into the system. This is a trivial task since Axis2 web services engine already supports pluggable transport architecture [12]. We used the NIO transport provided by Apache Synapse project [15] to enable non-blocking I/O in Axis2.

Since the frequency of subscription operations (e.g. add/remove Xpath or topic subscriptions) are significantly lower than publish operations (e.g. message receptions) rate, we focused on processing publish operations in parallel.

C. Parallel message delivering subsystem

In order to counter the problems involved with the serial message dissemination we have considered two parallelization strategies: topic based parallelization and consumer based parallelization. In the topic based parallelization several threads are assigned to a set of topics/Xpath expressions. These threads will deliver a particular message to consumers if the message matches to a topic/Xpath handled by each thread. However implementation of this becomes problematic due to two reasons:

1) If a message matches to more than one topic/Xpath expression, which were submitted by the same consumer but are handled by different threads, the system should decide which thread is going to deliver the message.

2) Suppose the system receives messages \(m_1\) and \(m_2\) consecutively at the time \(t_1, t_2\) where \(t_1 < t_2\) and \(m_1, m_2\) matches to topic/Xpath expression \(X_1\) and \(X_2\) respectively which are handled by different threads but submitted by the same consumer. In such a scenario the system should make sure that the consumer receives \(m_1, m_2\) in the order they arrived.

Solving above problems need sophisticated synchronization mechanisms. Therefore, this strategy was not implemented.

The other parallelization strategy is based on consumer end point references (EPRs). In this method, each consumer EPR is assigned a standing job and a message queue. The responsibility of these jobs is to deliver messages accumulated in queues. Note that standing jobs and queues are not created immediately upon receiving a subscription, but they are created only if messages are available for a particular consumer. Additionally once created, jobs will expire after a certain time period if there aren't any messages accumulated for relevant consumers. These two steps are precautionary measures to ensure that the system is not burdened with unnecessary idling jobs. The System supports two approaches to schedule standing jobs, which we term as dynamic and static.

D. Scheduling with a dynamic thread pool

A subtle difference is introduced to standing jobs a when a dynamic thread pool is used where the job will drain the entire message queue at once (Figure 2) rather than de-queuing one message at a time. This is to avoid locking overhead associated with repeated polling of the queue [9]. After delivering all drained messages, the job will again check the queue for newly arrived messages. If the queue is empty, the job block-waits for a configurable time period for messages to arrive. However, if messages don't arrive then the job will expire eventually, releasing all the resources associated with it.

Note that these jobs are scheduled in a dynamic thread pool that creates new threads as needed but will reuse previously constructed threads once they are available. Threads that have not been used for a certain time are terminated and removed from the pool [9].

In this method, jobs will not release the control of its thread as long as their associated queues have messages. Therefore, we expect the system to utilize more resources for consumers who receives more messages than others.

E. Scheduling with a Static thread pool

![Figure 2 message delivery based on dynamic thread pool](image2)

![Figure 3 Message deliveries using static thread pool](image3)
In this approach (figure 3), the number of threads in the pool are kept constant. Standing jobs also behave differently than earlier scheme where they only drain a certain number of messages at a time. Once drained messages are delivered, the job will pass the program control to the thread. Each thread will iterate over standing jobs assigned to it continuously. Note that both these numbers are exposed as configurations so that the system user may fine tune the values accordingly.

VI. EXPERIMENTAL RESULTS

A. Experimental setup

In this experiment we used a dual core 2.1 GHz machine, with 1Gb RAM, as the server for our publish/subscribe system. This relatively low configuration enabled us to load the software without loading the 100 mbps-Ethernet network. Statistics were generated by sending a high load of XML messages to match over a constant set of subscriptions. We measured the system’s average throughput and the average message round trip time continuously over each test run.

To begin with, we compared the throughput (Figure 4) for blocking and non-blocking I/O transports. Here, the measurements were taken at the stable state of the system. The average throughput was calculated by every two seconds time. The multiple XML data streams were used to send out messages while the system used 10 topics to filter them. Topics and data were chosen such that 4000 messages will match each topic, which intern provided 40,000 messages altogether. We used 200 subscribers which subscribed to each topic in an alternating fashion.

Then we performed the experiment for the same number of publishers, subscribers and messages setup for each serial, dynamic parallel and static parallel implementations of the system. Figure 5 depicts the throughput for each system and Figure 6 shows the round trip time.

B. Performance Comparisons

Figure 4 depicts the comparative performances of the system with and without NIO. It can be clearly seen that the NIO based transport greatly increases the throughput of the system. This is due to the fact that the NIO transport handles more concurrent connections than non-NIO transport while using minimum resources. Hence more publishers are able to publish messages simultaneously while the system is able to process more messages.

Figure 5 shows the throughput of each method against time. It can be seen clearly, the serial implementation has the worst performance, with respect to rate (throughput). On the other hand throughput of the static thread pool based parallelization shows a moderate level of performance. However, dynamic thread pool based method gives the highest performance.

Figure 6 shows the round trip time of these three implementations against time. The serial sender is incapable of delivering messages at the same rate as it receives them. The nearly linear growth of the round trip time for serial implementation is a result of this deficiency. Compared to the serial method, both parallel scenarios give a very low round trip time. For clarity, we have enlarged a portion of the diagram which shows the performance of the parallel algorithms.
VII. CONCLUSIONS

This paper focuses on the message delivery implementation of publish-subscribe system, which is one of the bottleneck prone part of publish/subscribe system design, and presents a few potential approaches and a detailed analysis of their comparative merit.

The observed problems are unreliable consumers, blocking IO, and serial message delivery. Among them, the problem of unreliable consumers is caused by consumers who subscribes and later fail, and consequently, the broker wastes a lot of resources trying to publish to those fail endpoints. We have proposed a blacklisting scheme to identify and manage these consumers hence avoiding performance degradations. Furthermore, we have used non-blocking IO based server as a solution for blocking IO. Due to inadequate performance of serial message delivery, message queue in the broker can grow easily when there is large number of consumers involved. Therefore, we have proposed two techniques to improve the message delivery by incorporating parallelism.

Furthermore, we have performed an empirical analysis of different approaches. As shown in our experiments, incorporating NIO transport lets users to achieve more performance. Even if both of the parallel message sending strategies performs better than serial sending, each method behaves differently in terms of throughput and the roundtrip time. Therefore it’s advisable to adopt the message sending strategy based on dynamic thread pool when the system throughput is more important than the message round trip time. Other parallel strategy can be adopted if the message round trip time is considered important compared to the throughput.

In conclusion, we have identified several problems related to message delivery in a publish/subscribe systems, and proposed several potential solutions to those problems. Furthermore, we have established their pros-and cons through an empirical analysis. We believe these results will be useful for designers of publish/subscribe systems.

VIII. FUTURE WORK

A. Black listing scheme

The blacklisting scheme can be further improved by determining the black listed time period and threshold count in a suitable manner. This can be done by adjusting them based on the failure type, the consumer characteristics, the system load etc. After introducing above changes we hope to analyze the system’s performance in a production environment to access the usability.

B. Improved static executor pool

In the current implementation of the static thread pool based parallelization strategy, newly created standing jobs are assigned to threads in a round robin manner. But the utilization of each thread is governed by factors such as number of standing jobs, message size, network delays between consumers and the system etc., causing thread underutilization. Therefore, it’s desirable to determine a method to distribute the standing jobs in a way to ensure equal thread utilization.

An interesting parallelization strategy has been introduced in [10] which is much similar to our static implementation, but incorporates a work stealing mechanism where idling threads fetch work from busy threads, which ensures proper thread utilization. This method can be introduced to our static parallel implementation as well.

Alternatively newly created standing jobs can be distributed over the static thread pool based on thread idle time. Thus, allowing the idletest thread to acquire the job. This concept will keep each and every thread equally busy.

Finally, we hope to access the performance of two parallel delivery mechanisms by varying the number of consumers.

REFERENCES


