A NEW PERFORMANCE OPTIMIZATION STRATEGY FOR JAVA MESSAGE SERVICE SYSTEM

Xiangfeng Guo, Xiaoning Ding, Hua Zhong, Jing Li
Institute of Software, Chinese Academy of Sciences, Beijing, China

Keywords: Java Message Service, Performance Optimization.

Abstract: Well suited to the loosely coupled nature of distributed interaction, message oriented middleware has been applied in many distributed applications. Efficiently transmitting messages with reliability is a key feature of message oriented middleware. Due to the necessary persistence facilities, the performance of transmitting is subject greatly to the persistence action. The Openness of Java platform has made the systems conforming to Java Message Service Specification supported widely. In these applications, many consumers get messages periodically. We bring forward a new efficient strategy using different persistence methods with different kinds of messages, which improves system performance greatly. The strategy also utilizes daemon threads to reduce its influence to the system. The strategy has been implemented in our Java Message Service conformed system, ONCEAS MQ.

1 INTRODUCTION

Message oriented middleware was used widely for its loosely coupled nature. It provides three decoupling dimensions (Eugster et al., 2003): time decoupling, space decoupling and control decoupling. Many industrial specifications including CORBA Notification Service (OMG, 2002) and Java Message Service (JMS) (Sun Microsystems, 2002) have been proposed due to the population of message oriented middleware. Among the specifications, JMS Specification has been supported widely by industrial products due to the popularity of Java Platform. There are lots of famous products conforming to JMS specification, including Websphere MQ (IBM, 2004), Weblogic (BEA Systems, 2003) and JBossMQ (JBoss Group, 2004).

JMS provides two kinds of communication paradigms: PointToPoint and Publish/Subscribe. JMS summarizing message queue systems and publish/subscribe systems provides wealthy operations and related semantics for us to build powerful applications.

JMS provides two persistence modes to transmit messages: PERSISTENT and NON_PERSISTENT. With NON_PERSISTENT mode the message may be lost due to system crash, but we can get a better throughput. While with PERSISTENT mode, each message will be logged into persistence store for reliability and will not be lost on system failures.

Many crucial enterprise applications, such as online stock trade system and working flow system, require high reliability. Persistence mechanism is employed to implement reliability, which usually logs messages into persistence store based on database or file system. Persistence operations are usually costly because of involved database or file operations. Therefore these operations influence system performance greatly. Using message cache is an effective approach to improve system performance.

We introduce subscriptions with additional information indicates whether the message consumer will get required messages periodically. In our approach, these messages will be prefetched to the cache. We use daemon thread to do the prefetching tasks to reduce the influence of the prefetching action.

The rest of the paper is organized as follows: section 2 introduces the background and the performance problem. Section 3 discusses the optimization strategy and implementation. Section 4 gives our evaluation. Section 5 reviews the related works. And we conclude the paper in the last section.

Guo X., Ding X., Zhong H. and Li J. (2006).
A NEW PERFORMANCE OPTIMIZATION STRATEGY FOR JAVA MESSAGE SERVICE SYSTEM.
In *Proceedings of the Eighth International Conference on Enterprise Information Systems - ISAS*, pages 520-523
DOI: 10.5220/0002498005200523

2 BACKGROUND

2.1 OnceAS MQ

OnceAS MQ is a message system conforms to JMS specification v1.1. It has several advanced features including:

– Supporting both PointToPoint and Publish/Subscribe communication paradigms.
– Supporting multiple communication protocols.
– Supporting multiple persistence policies, including policy based on database and policy on file system.

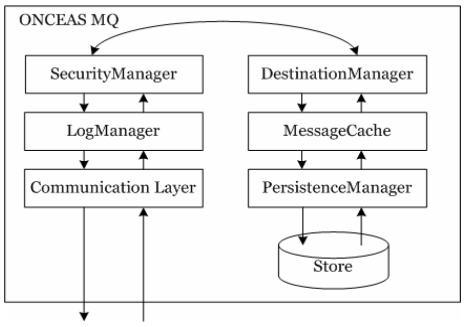

Figure 1: Architecture of OnceAS MQ.

The architecture of OnceAS MQ is illustrated in Fig. 1. As we can observe from Fig. 1, OnceAS MQ was mainly composed by several components. CommunicationManager is responsible for the communication between clients and server; LogManager records all of the communication events and message processing events; SecurityManger is responsible for security issues, such as authentication and authorization; DestinationManager manages all destinations in the system, including Queues and Topics; MessageCache is the caching component for improving performance, which will be introduced later in detail; PersistenceManager takes charge of persistence issues.

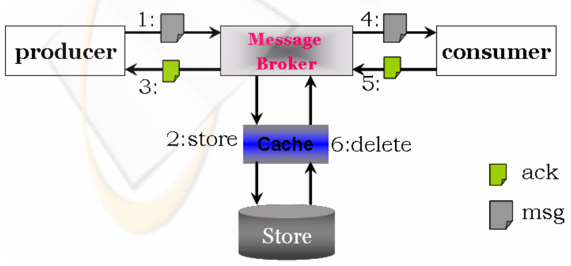

Figure 2: Architecture of OnceAS MQ with a cache.

2.2 Reliability Guarantee

To achieve a reliable message transferring, we need combine both a reasonable persistence mechanism and a reasonable transferring protocol. In ONCEAS MQ, following protocol shown in Fig. 2 was employed to guarantee the reliability of transferring:

1. The message producer sends its message to the message broker.
2. The message was stored into a persistence media.
3. The message broker sends an acknowledge message to the message producer.
4. The consumer fetches the message from the message broker.
5. The consumer sends an acknowledge message to the message broker.
6. The message broker deletes the message from the persistence media.

We import the caching service for a better performance, which was managed by MessageCache in OnceAS MQ. The cache is stored in RAM for a higher speed than in persistence store. However, RAM is a scarce resource in system. So some policies must be applied to utilize it more effectively.

3 OPTIMIZATION STRATEGY AND IMPLEMENTATION

In many applications based on message systems, for example online stock trade system, data sampling system and weather information system, message consumers get messages periodically. But message system does not know whether a message consumer will get messages periodically or not because its submitted subscription has no information about that. Our System allows message consumers to declare whether they will get specific messages periodically. Some applications may want message system to push messages to them. The push manner requires message consumer to register a *MessageListener* and usually requires message system to maintain a connection between message system and message consumer. This manner will be costly and the applications usually pull messages from message system to get a better performance.

We mainly focus on performance optimization of the pull manner. We also find that when a message consumer gets a message it usually will also get some related messages. We introduce extended subscription to improve performance, which is with additional information indicates whether required messages will be consumed periodically.

Message system decides whether a message will be got from the system periodically through the following two ways:

– Consumers declare the feature explicitly in their subscriptions. This can be done by adding extra information to the parameter of *createConsumer()* or *createDurableSubscriber()* operation.
– System can use a heuristic method to predict consumers' periodically getting actions. This is useful when there are a lot of periodical getting actions but consumers have not declared their periodical actions.

Our approach is to prefetch periodical messages into the message cache just before the time consumers get them. We must decide which messages to be prefetched and when do the prefetching tasks, as we will discuss below.

3.1 Message Prefetching

Due to the limited size of the cache, we should only cache messages which are most likely recently used. Our approach is based on the LRU algorithm considering the periodical getting actions of consumers and message relations.

When a message producer sends a message to message system, the message will be logged directly into the persistence store if it will not be used recently according to subscription information. Otherwise the message will be put into the cache. If the cache is full, a message less important in the cache will be deleted from the cache so that the new message can be put into the cache. The importance of a message is decided by the following factors: message size, message priority, reside time in the cache and user defined factors. We can use administration tool to configure these factors.

Suppose a message will be got at time *T* according to subscription information, we must prefetch the message to the cache before time *T*. Related messages will also be prefetched to the cache because they may also be got from the system. We must decide the time to begin to prefetch messages. If we prefetch messages too early, the cache will be used unnecessarily. We must estimate the time $T_{prefetch}$, time used in getting messages from the persistence store to the cache. We can begin the prefetching tasks at time $T - T_{prefetch}$ so that messages will not reside in the cache unnecessarily.

But doing this leads to another problem. At time $T - T_{prefetch}$ there may be a lots of messages need to be prefetched. This will cause the system too busy and subsequently decrease system performance. Therefore we should do the prefetching tasks without causing the system too busy. We can do prefetch messages at a time before time $T - T_{prefetch}$. The performance is also influenced due to doing all the prefetching tasks as one unit as shown in Fig. 3.

To reduce the influence of prefetching to other system activities, the prefetching tasks is divided into many pieces in our approach, as shown in Fig. 3.

We also use daemon threads to do the prefetching tasks so that the prefetching action can be done at time system is not busy. However, we can not predicate when daemon thread will run. Thus we can not guarantee that the prefetching tasks will be finished before consumer gets the messages.

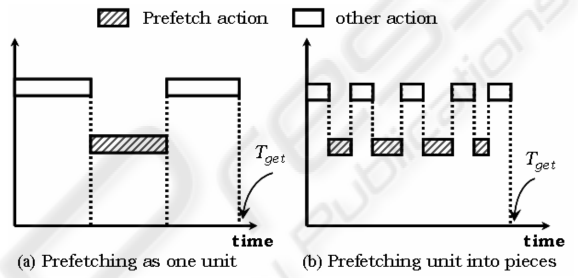

Figure 3: Diffrent prefetching methods.

In ONCEAS MQ, we proposed a new approach, which does the prefetching tasks in system idle time using daemon thread to some extent and can also guarantee the tasks will be finished before the time consumer gets messages from the system.

We divide the tasks into *n* parts, and all the work must be done in a time interval $T_{prefetch}$ before time *T*, the time consumer gets messages from the system. We delegate daemon thread to do the tasks, but if the daemon thread hasn't finished *x%* of the tasks when the time elapsed *x%*, the system will do the prefetching tasks using another non daemon thread until *x%* of all the work finished.

3.2 Average Occupying Time in the Cache

When a message has been sent to the system, whether put it to the cache or just log it into the persistence store is decided by the interval between the time the message has been sent to the system and the time the consumer gets it out from the system. We must decide the value of the interval. In our approach we define the value as the average occupying time of a message in the cache. Suppose the cache size is *S*, in a time interval of *T*, *N* messages have been got out. The average occupying time of a message is *T*S/N*. If a message will occupy the cache no longer than the interval value it will be put into the cache. Otherwise we just log it into the persistence store.

4 EVALUATION

We deploy an online stock trade application on ONCEAS MQ system. In the online stock trade application, message producers will send stock information wrapped in messages to the system while message consumers will subscribe messages they interest in.

We give a simulating application that has 30 message consumers interested in different information of 100 stocks. The consumers get messages every 15 minutes on different time. The evaluation result is shown in Fig. 4.

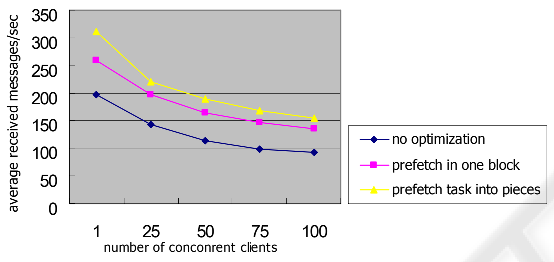

Figure 4: Evaluation Results using different methods.

From Fig. 4, we can see that by applying subscriptions with periodical information, we got an average performance improvement of about 30% for the consumers. By dividing prefetching tasks into pieces, we get another performance improvement of about 11% than doing the prefetching tasks as one unit.

5 RELATED WORK

Considerable research has been done in using cache to improve system performance in many areas, including general purpose application level cache (Iyengar, 1997), web cache (Li et al., 2000) and caching performance of Java platform (Rajan, 2002). Java Temporary Cache (JCache) (Sun Microsystems, 2001) has been proposed by Oracle and provides a standard set of APIs and semantics that are the basis for most caching behaviours. But there is few work considering how to improve performance of applications based on messaging systems. JBoss MQ has implemented a message cache simply based on the LRU algorithm with no consideration of characteristics of the messaging system. BEA Weblogic and IBM Websphere MQ have no consideration of characteristics of subscriptions. Our approach allows the user to use subscriptions with periodical information and apply corresponding optimization strategy to improve system performance.

6 CONCLUSION

This paper proposed optimization strategy to improve performance of message system with message cache. We extend subscriptions with periodical information and prefetch messages to the cache efficiently. Our approach does the prefetching tasks using daemon thread as likely as possible. To reduce the influence of the prefetching action to the system, we split prefetching tasks into pieces and do them at different time.

In future research, we will incorporate the features of Java Virtue Machine into our strategy to achieve further improvement. Caching Optimizations in distributed environment will be studied. We will also focus on tuning of difference parameters of the caching system, such as cache size, number of pieces and number of threads.

REFERENCES

BEA Systems, 2003. *Programming WebLogic JMS.* http://e-docs.bea.com/wls/docs81/jms/.

Eugster, P., Felber, P., Guerraoui, R. and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Computing Surveys,* 35(2), pages 114 – 131, Jun. 2003.

IBM, 2004. *Websphere MQ.* http://www.ibm.com/software/ts/mqseries/.

Iyengar, A. Design and Performance of a General-Purpose Software Cache. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems,* December 1997.

JBoss Group, 2004. *JBoss MQ.* http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossMQ.

Li Fan, Pei Cao, Jussara Almeida, Andrei Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3), pages 281 – 293, 2000.

OMG, 2002. *CORBA Notification Service specification version 1.0.1.* http://www.omg.org/corba.

Rajan, A., 2002. A Study of Cache Performance in Java Virtual Machines, *Master's Thesis,* University of Texas at Austin.

Sun Microsystems, 2001. *JCache: Java Temporary Caching API.* http://www.jcp.org/en/jsr/detail?id=107.

Sun Microsystems, 2002. *Java Message Service (JMS) API Specification.* http://java.sun.com/products/jms/.