A Resilient Message Queuing Middleware

Mohammad Reza Selim, Yuichi Goto, and Jingde Cheng
Department of Information and Computer Sciences
Saitama University, Saitama, 338-8570, Japan

Abstract

Message Queuing Middlewares (MQMs) are gaining more and more attention in large enterprises for building highly available asynchronous messaging systems and for integrating heterogeneous applications. However, currently available MQMs consider underlying networks as static. Therefore, in case of node failures or a disaster, either they have to suffer long term service loss or they need to install a lot of extra resources to ensure that no such failures cause service loss. They also require large administrative overhead as the network is managed manually. Besides, as store and forward method is used, reliable delivery of messages suffers much network delay and generates large amount of traffics. Current MQMs are problematic especially if the network contains a large number of nodes. In this paper, we propose a design of a Pastry based middleware which can provide an asynchronous, reliable and in-order delivery service while ensuring no long term service loss in case of failures or disasters. Such services can be provided with minimum deployment cost. Our simulation based evaluation shows that we can provide such services in a network of large number of nodes while generating less traffic and requiring minimum administrative overhead.

Keywords: *Asynchronous Messaging, Availability, In Order Delivery, Message Queuing Middleware, Reliability.*

1. INTRODUCTION

In today's business environment applications need to be connected loosely to accept continuously changing business roles. They also often need to communicate with each other in a point to point/multi-point basis. The purpose of *Message Queuing Middlewares* (MQMs) is to enable applications (also called *clients* or *programs*) to communicate across a network, without having a private, dedicated, logical connection to link them [1, 2]. Applications communicate indirectly by putting messages on message queues of the middleware, and by taking messages from the queues [1, 2]. MQMs are usually used when the communicating applications need to execute independently and concurrently without waiting for one another to reply, when the users are often disconnected, for

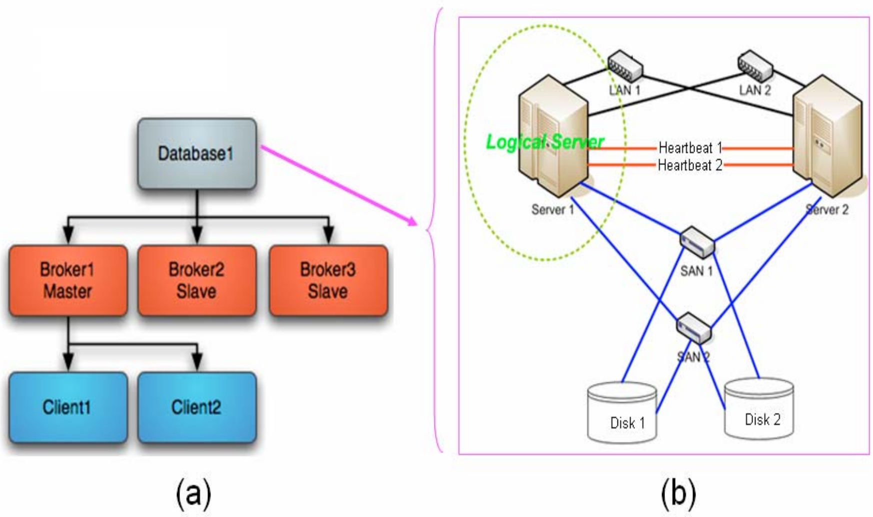

Figure 1: (a) HA master/slave clustering of brokers (Source: [5]). (b) HA clustering of database servers (Source: [4]).

example travelling sales men, etc. The queues may be distributed across a network. The applications request a *Queue Manager* [1] running in a middleware node to route the message to the destination queue. The queue managers are called *Brokers* or in some middlewares, e.g., in Microsoft Message Queuing (MSMQ) the *Routing Servers* [2]. In a large enterprise level deployment each site contains at least a broker or a *High Availability* (HA) *cluster* [4] of brokers and the applications are connected to that broker or a broker in the cluster [2, 3, 5].

Considering single broker per site, if the broker of a site fails or taken offline for periodic maintenance or upgrade, the applications does not have any broker to connect to. It even does not have any way to fail over to a remote broker. Thus it suffers a *long term* service unavailability [3].

To avoid this loss, clustering is used. There are several approaches of clustering. Fig. 1(a) shows a popular approach called HA *Master/Slave* broker cluster [5]. As we can see the brokers of a cluster keep messages and configuration files in a shared database. All the applications of a site are connected to a broker called *master broker* who holds the lock of the shared database. Other brokers in the cluster continuously try to get the lock of the database. If the master fails, one of the slaves gets the lock and becomes the master. If the master gets back, it joins as a slave. Thus if a broker fails, applications can failover to another broker. However, please note that a broker can work only if the database is available. To ensure that a database does not fail, another HA cluster of database servers is necessary as shown in Fig. 1(b). Note that use of database cluster can be avoided if data is synchronously replicated to all the brokers of the cluster. However, this approach has significant overhead as replication is done for every incoming and outgoing message. Again, what will happen if a *disaster* occurs and all the brokers and database servers are destroyed? We, again, need to invest for disaster recovery.

Therefore, one problem of *currently available MQMs* (we also call *traditional MQMs* or simply *MQMs*) is that there is no *cost effective* way to ensure that the service will not be stopped for a period of failure of a broker or during periodic maintenance or upgrade.

MQMs' responsibility is to transfer the messages up to the destination queue. Then the application's responsibility is to get the messages from the destination queue. As *store and forward* [2, 3, 5] method is used, messages are stored at each broker from the source application to the destination application. After storing at a broker, an acknowledgement is sent to the sending machine.

MQMs provide both *transactional* and *non-transactional* messaging services [2]. If a set of messages is indicated as transactional they are delivered to the destination queue *exactly once* and in the *order* they are received. Non-transactional messagings are classified into two types, *express* and *recoverable*. Express messages are not stored in persistent storage but only in the main memory. Therefore, they are very fast but they are supposed to be lost if content of the memory is lost (it can be caused by, OS crash, a reset etc). On the other hand recoverable messages are stored in persistent storage so that they can be recovered in case of failure or cash. In contrast to the transactional messages, the non-transactional messages are not guaranteed in order or exactly once delivery semantics; however they are faster and have less overhead than the transactional messages.

Both transactional and non-transactional messages can be set to have reliability property which means that after the messages reach to the destination queue, an acknowledgement will be sent to the source site. Another acknowledgement will be sent after the message is consumed from the queue. These acknowledgements will follow the reverse path from destination to the source queue. Therefore, the sending application can be sure that a message has been reached to the receiving application. Note that reliable messages have much overhead as two acknowledgements are necessary in addition to acknowledgement of receipt of a message at each intermediate broker.

We have mainly surveyed routing mechanism used in MSMQ. It uses an efficient routing algorithm to transfer a message from source queue to the destination queue. The routing algorithm used in MSMQ is called *Binary Reliable Message Routing Algorithm* (MS-MQBR) [7]. The enterprise is considered as a set of sites. Each site has link to one or more neighboring sites whom it can communicate directly. These links are called *routing links* who identify neighboring MSMQ sites. The administrator sets a cost to each routing link. This cost represents how expensive it is to transfer messages directly between the two sites.

To build the routing table, each broker considers the enterprise as a directed graph $G = (S, E)$ where S = *set of vertices*, i.e., the sites and E = *set of directed non-negative weighted edges*. MSMQ then uses Dijkstra's algorithm [10] to find *least-cost paths* to each destination site by finding a *spanning tree* [10] that covers the entire graph. The algorithm populates the routing table from the built spanning tree. The routing table contains two fields {*DestinationSiteID, NextHopSiteId*}[7]. When sending a message to a queue, to use the routing table, the broker must know the ID (called Global Unique Identifier or GUID) of the site where the destination queue resides. MSMQ must use another service running in the same site called the *Active Directory Service* [2] provided by the Windows Servers. Active directory maintains all the queues/objects created in the whole enterprise (not only those created in the same site) and the GUIDs of the sites where they have been created.

To make the algorithm work, in addition to the directory services several data structures are needed. A table called *SiteRecordTable* of size $O(N)$, where N is the number of sites, containing all the site information. A table called *RoutingLinkRecordTable* containing cost information of all the site links. If the average number of direct links from one site is f, the size of this table is $f \times N$. A table called *MachineRecordTable* containing the node/machine (i.e., site gate, routing servers, connected networks etc.) information of all the sites. The size of the table is $O(N)$.

The routing performance depends on how accurately the administrator estimated the link costs, how many direct links for each site have been inserted into the *RoutingLinkRecordTable* table. For reasonable values of these variables, the routing efficiency should be very good.

However, the problem lies elsewhere. Although the routing performance is better, as the message is stored in each intermediate hops, it poses a significant amount of delay to the non-express messages. It also generates much traffic. Besides, all the aforementioned data structures must be maintained manually by the site administrators. This is an error-prone and time consuming job requiring highest administrative overhead. Thus they are especially problematic if the network contains huge number of nodes.

In this paper, we present a design of a new middleware based on Pastry p2p protocol [11], called ***Pastry Based Message Queuing Middleware (PBM)***, to provide an asynchronous point to point messaging service having reliable, exactly once and in-order delivery guarantee while eliminating the aforementioned limitations. In other words, the proposed middleware will have low traffic overhead and deployment cost but will not cause any long term service loss to the applications. It also eliminates administrative overhead while maintaining reasonable routing performance. Our middleware currently supports only point-to-point communication. However, multi-cast communication semantics can be built over this point-to-point service. The novelty of our work is that we first provide an in order and reliable messaging services over a structured p2p network.

The rest of the paper is organized as follows. Section 2 presents the design details of our proposed PBM. We analyze the messaging performance and traffic generation of both PBM and MQM in Section 3. Section 4 evaluates PBM with respect to MSMQ. Finally before conclusion in section 6, we present related work in section 5.

2. DESIGN OF THE PASTRY BASED MQM

As we have already described, the main problem of non-cluster deployment of an MQM is that if it fails, brokers of other site can not take over the responsibility. This lack of dynamism is solved by using Pastry structured p2p network. As Pastry network has self-managing characteristics [11], it requires no administrative overhead to maintain the routing information. Besides, as one broker uses the resource of another broker (in another site) to replicate the stored messages to recover from failures, it does not need any cluster. Thus it reduces the cost to deploy and mange a cluster.

Among a number of structured p2p protocols, we use Pastry because of two reasons: it is very flexible, e.g., the average number of hops of a message can be adjusted by varying several parameters and, most importantly, it provides a proximity routing which facilitates an approximate mapping between physical and overlay network [11].

Throughout our discussion we assume that all the channels are unreliable (and hence faster). Unlike MQMs, in our system a message is always destined to an application not to a queue. Queue is managed implicitly. This relieves the application developers from managing middleware queues.

2.1. Basic Approach

Our approach is very simple. We let the brokers form a Pastry based p2p network as shown in Fig 2. Each broker is assigned a 128 bit (hashed) ID called *nodeId* which is obtained by applying a hash function *H* on the IP address/public key of the broker. Similarly, we use the same hash function on names/address/public keys of an application, say A, to get its hashed *key H(A)*. An application gets connected to the broker whose *nodeId* is numerically closest to its *key* to get messaging services from the network of brokers. We call this broker the *responsible broker* of A and represent as *resp(H(A))*. Unlike traditional MQMs, responsible broker not necessarily resides in the same site as of the application it is responsible for.

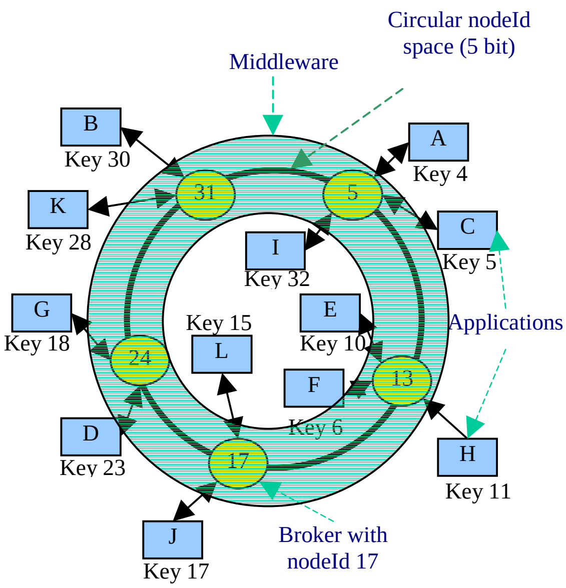

Fig. 2: Pastry based MQM and applications

2.2. Types of Messages Supported

Like traditional MQMs, our proposed PBM supports both *non-transactional* and *transactional* messages. Transactional messages are by default reliable and delivery order is maintained. Therefore, they must be acknowledged. If a message is set as non-transactional but reliable, they must also be acknowledged. But unreliable messages need not to be acknowledged. All messages, either transactional or non-transactional, in our system are not necessarily stored in persistent storage. As we will see such an approach is not suitable for our system. If the receiving application is offline all messages except the *express* messages are replicated by the destination broker. Unlike MQMs our middleware has not adopted the *atomicity* property of a transaction of several messages (other than only single message) yet but we hope to adopt it in our future work. Table 1 shows various types of messages and their properties.

TABLE 1: SUPPORTED MESSAGE TYPES AND THEIR PROPERTIES

Type	Reliability	Properties
Express	Reliable	Never replicated (by the destination broker) but acknowledged
	Unreliable	Neither replicated nor acknowledged
Recoverable	Reliable	Replicated and acknowledged
	Unreliable	Replicated but not acknowledged
Transactional	Reliable by default	Replicated, acknowledged and delivered in order

2.3. Message Transfer

Unlike MQMs, we avoid store-and-forward approach for it's inefficiency. By *storing*, even if we mean to store not in persistence storage rather in main memory (more specifically in a Main-Memory Database called MMDB), it will not be very efficient in PBM. As in our approach if a broker fails, another broker takes over the responsibility immediately; if we want to use store-and-forward approach we have to store in one broker and replicate it in several other brokers over the network. This will cause a severe delivery delay specially if this process is repeated, like MQM, in all intermediate brokers in the path of a message.

Therefore, our approach of transferring a message is as follows. When an application A wants to send a message to another application B, it sends the message to the responsible broker *resp(H(A))*, i.e., the broker whose ID is numerically closest to the key *H(A)* of the application. We call this broker the *source broker* of the message. The source broker keeps a copy of the message in the main memory to resend later if necessary. The responsible broker then sets the destination field as *H(B)* and sends the message. The message then reaches, may be via some intermediate brokers, to the destination broker *resp(H(B))* who is the numerically closest node of *H(B)*. The destination broker now checks the status information to know if application B is online or not. If it is online, it sends the message to it. After receiving the message application B will send an acknowledgement back to the responsible broker. This acknowledgement will now be forwarded to the source application A through *resp(H(A)).*

However, if B is not ready to take the message or if it is currently offline, the message is replicated to *K-1* number of numerically closest brokers of *resp(H(B))* and stored in the main memory based queue of *resp(H(B))*. Only after this replication and storing operation is confirmed, an acknowledgement is sent to *resp(H(A))*. Please note that all storing and replicating operations are performed on main memory before sending an acknowledgement. However, after sending acknowledgement, the memory based queues can be stored in persistent storage which may be necessary to save the main memory space but not essential for recovery or other purposes.

Messaging in our middleware differs with that in traditional MQMs. In MQMs messages are stored in every intermediate broker but in case of PBM it is not stored in intermediate brokers other than source and destination brokers because unlike MQM, an acknowledgement of a message need not follow a reverse path.

In PBM communication between application and broker is, in almost all the cases, inter-site because as a hash based approach is used, an application not necessarily resides in the same site of its responsible broker. Therefore, an application's messages may need to be sent on the first hop to a broker in other site. This may cause security risks. However, as we consider that MQM is deployed in an enterprise boundary, a broker in another site belongs to the same enterprise causing less risk. Nevertheless a security mechanism must be adopted. This is subject of our ongoing research. There are some existing work on similar issue, e.g., PAST[15].

2.4. In-order Delivery and Duplication Elimination

In PBM transactional messages are sent in-order and are not duplicated at the destination application. In our proposed system, if a message is transactional, the source application will not send a second transactional message until it receive the acknowledgement either from the destination application or from the destination broker. This slightly differs with traditional MQMs where an application can send several transactional messages together to the source broker. This appears to be a faster process. But the fact is that the source broker will send those messages to the destination broker sequentially like that in PBM [3]. The source application can not be sure about the fate of a transactional message until an acknowledgement comes from the destination broker stating that the messages have been delivered to the destination broker/application. Therefore, this difference between MQM and our middleware is not an issue if we consider *application-to-application* or *application–to-destination broker* delivery because in such a case MQM should not

```
processMsg(m) //runs in broker
   if (queue[m.dest] = NULL) queue[m.dest].insert(m)
and return
   maxMsgId=queue[m.dest].getMaxMsgId(m.src,
m.dest, isTransactional=true)
   if (m.isTransactional)
      if (m.id > maxMsgId) queue[m.dest].insert(m)
   else if ((m.id != maxMsgId)) queue[m.dest].insert(m)
end

processMsg(m) //runs in application
   maxMsgId = lastAcceptedMsgId[m.dest])
   if (m.isTransactional)
      if (m.id > maxMsgId)
         Accept the msg
         lastAcceptedMsgId[m.dest]) = maxMsgId
    else accept the message
end
```

Fig. 3: In-order delivery and duplication elimination algorithm

be faster than our middleware.

About non-transactional messages, we do not put any restriction like transactional messages. It can send a message before getting a reply of the previous one, as the non-transactional messages need not maintain any order.

Duplication elimination and in-order delivery of messages work together in our middleware. We define in order delivery as follows: if an application A sends two transactional messages m_1 and m_2 at times t_{s1} and t_{s2} respectively to the same application B where the messages m_1 and m_2 are ***accepted*** at times t_{a1} and t_{a2} respectively; if $t_{s2} > t_{s1}$, then $t_{a2} > t_{a1}$ must be true. Please note that, PBM (like MQM) does not ensure in-order delivery if the sources or the destinations of two transactional messages are different. In applications, an out of order transactional message, which must be a duplicate of a previously received message, is discarded.

To ensure in order delivery and duplication elimination, each message is tagged with a *message ID* by the source application. The message ID need not to be consecutive but must be in increasing order. In other words, if message m_1 and m_2 with IDs i_1 and i_2 are generated by an application at times t_1 and t_2 respectively; if $t_2 > t_1$ then $i_2 > i_1$ must be true but it not necessarily be true that $i_2 = i_1+1$. The following rule must be satisfied by each application to ensure an in order delivery.

In Order Delivery Assurance Rule: *If the ID of the last accepted transactional message from an application is i, accept a received transactional message sent by that application only if the ID of the message is greater than i.*

To follow this rule, each application must remember the ID of the last received transactional message from each application. This requires a data structure of maximum size equal to the number of applications in the system.

However, a broker does not maintain such a data structure; therefore it may accept an expired transactional message. When a broker receives a transactional message of id i_1 whose source is *A* and destination is *B*, it checks the queue maintained for the application *B*. If the queue is empty or there is a message in the queue with id $i_2 < i_1$ for the same source and destination, it accepts and insert the message in the queue. Therefore, if the queue is empty, an old message which is already delivered may be accepted and inserted into the queue. An application may therefore receive an old (and hence duplicated) transactional message. However, as the application needs to maintain the stated data structure, such old transactional messages are not accepted rather discarded. Fig. 3 shows the algorithm used for this purpose.

Note that in traditional MQM based systems an application is not free from running a duplication elimination algorithm also. MQM confirms the exactly once delivery of a transactional message up to the destination queue (not up to the application). To get a message from the queue which is in a different machine, the application need to run a duplication elimination algorithm if it wants to get the message exactly once.

Please note that once a message, which was replicated to *K-1* closest brokers, is delivered to the receiving application, the responsibility of the destination broker is to try to delete the replicas from the *K-1* closest brokers. For this purpose, the destination broker sends a replica deletion message to those *K-1* brokers. These messages are sent as *unreliable express message*. If this message does not reach to a broker containing the replica, it can not delete it. Therefore, an inconsistency may occur among the queues maintained by the K-1 brokers. What this loose consistency can do is to cause an expired message to send to the application. As we have already seen that this case is handled by the duplication elimination algorithm.

2.5. Message Reliability

If a reliable message is delivered to the destination application, it sends the acknowledgement which is received by the sending application. However, if the message can not be delivered to the receiving application, the destination broker replicates and stores the message and sends an acknowledgement. After receiving the acknowledgement the sending application can understand that the message has only been kept by the destination broker but has not been delivered to the receiving application yet. After the message is delivered to the application, the destination broker sends a *delivery confirmation* message to the sending application. This message is treated as *unreliable express message*. Therefore, if the sending application is not online, the message is stored (but not replicated) by the responsible broker of the sending application. If the sending application does not receive any delivery confirmation, it can send the message again. This surely can cause duplication of a message in the receiving application. But as each application runs a duplication elimination program, duplicated messages are automatically discarded.

2.6. Handling of Application Failures and Arrivals

If an application leaves the system, the responsible broker does not need to know it immediately. When the broker sends a message to the application and gets no reply it understands that the application has left the system. The broker then starts replicating and storing messages on behalf of that application until the application reconnects.

If an application reconnects, the stored messages are delivered to the application first. We call this operation *stored message delivery*. Before this operation is finished, the broker needs to take care to initiate any normal transactional *message delivery* operation because it may cause unordered delivery of messages. Suppose *stored message delivery* operation contains a transactional message m_1 and before the operation is finished another transactional message m_2 is attempted to deliver to the same application (i.e., $m_1.destination = m_2.destination$) under a normal *delivery* operation. Note that if $m_1.source = m_2.source$, $m_2.id > m_1.id$ as m_2 has generated after m_1. In this case if m_2 is propagated before m_1, m_2 will be accepted but m_1 will not according to the *in order delivery assurance rule*. The ultimate result is unordered delivery of messages. This situation, although rare, can occur because, unlike MQM, we are not storing every message in the queue. We avoid such concurrent situations by prohibiting the normal message delivery operation that fulfills the above constraints during a *stored message delivery* operation. Fig. 4 shows a simplified algorithm to avoid such situation.

```
deliver(m , key)  //m –message, key – destination
application
   if (isContinuingSMD(key) //is stored message delivery
                              //operation is going on to key?
                 if (!checkSMD(m.source, isTrans=true,
m.id))
                               send m to key
                 else do not send until
!isContinuingSMD(key)
         else send m to key
end
```

Fig. 4: Algorithm to avoid unordered delivery due to concurrent delivery operations

2.7. Handling of Broker Failures and Arrivals

If a broker fails or a new broker joins, the leaf set of some other brokers is changed. If the leaf set of a broker is changed, a function named *update(nodeId,joined)* is called at the upper layer of the same broker. The current broker then checks if the broker (with ID *nodeId*) that joined/left belongs to or were belonged to the set of *K-1* closest nodes called *replicaSet*. If so and if it has left, the current broker sends a message to it to delete the data that it kept as replicas on behalf of current node (assuming that it has left the replica set but not the network). On the other hand, if it has joined, the current broker sends the joining node all the necessary data that need to be replicated. The data includes the transactional and recoverable messages and other information (e.g. application profile) the current broker stores on behalf of the applications it is responsible for. The data that is necessary to send can be bundled together before sending to reduce the protocol overhead. We call this operation *stored data replication.* Fig.5(a) shows a simplified algorithm of *update()* function.

However, there is a *"loss of order"* issue here. This may cause due to concurrent operations. Suppose the current broker is sending a transactional message m_1 as part of its normal *replication* operation and before it is completed, a *stored data replication* operation is initiated which sends another transactional message m_2 as part of its operation. Note that $m_1.id > m_2.id$ as m_2 has already been acknowledged by the broker. If the destination (i.e., the joining broker) of both operations is same and if $m_1.source = m_2.source$ and $m_1.destination = m_2.destination$ then there is a probability that m_1 will reach first before m_2. In such a case m_2 will not be accepted (although m_1 has already been accepted before m_2) by the joining broker (we have already discussed the reason in subsection D of this section). Therefore, the queue of the joining broker will not contain m_2. The ultimate result may be that m_1 is delivered to the destination application before m_2, causing a loss of order. We avoid this by prohibiting such situations to occur concurrently. Note that probability of occurring such situation is very little as a lot of constraints are related to it. Fig.5(b) shows a simplified algorithm to avoid such situation.

```
update(nodeId, joined)
   if (!joined AND wasInReplicaSet(nodeId))
      send a message to nodeId to delete all replica
   kept for thisNode
         else if (!joined AND isInReplicaSet(nodeId))
                  send necessary replica to nodeId
end
```

(a)

```
replicate(m , nodeId )  //m –message, nodeId –
destination of replica
   if (isContinuingSDR(nodeId) //is stored data
   replication
                              //operation is going on to
                           nodeId?
                  if (!checkSDR(m.source, m.destination,
isTrans=true, m.id))
                              send m to nodeId for
replication
                  else do not send until
!isContinuingSDR(nodeId)
         else send m to nodeId for replication
end
```

(b)

Fig. 5: (a) Algorithm to handle broker join and leaving. (b) Algorithm to avoid unordered delivery due to concurrent replication operations

Each broker sends a periodic message to the alive applications it is responsible for to inform its presence. If an application does not receive a periodic message it understands that its broker has failed or left. It then sends a lookup message to a known broker to know who is responsible for it. After getting the address of the new broker it can connect to it and can resume its operation. Thus within a little time an application can failover to another broker. In case of MQM, if a broker fails and there is no cluster member to takeover, the applications have to wait until broker(s) of that site is fixed [3].

3. MESSAGING PERFORMANCE AND TRAFFIC ANALYSIS

We analyze the performance and the generated traffic of PBM and MQM in this section. We have not considered the failure of a broker here because of two reasons. First, in MQM if broker(s) of a site fails, the applications of that site must have to wait for services until the failed broker(s) resumes.

While in PBM, there is always some broker to provide services to the applications. Besides the routing links need to be updated manually in MQM compared to automatic update in PBM. These incompatible natures between MQM and PBM make comparison somewhat illogical. The second reason is that as message queuing systems are deployed in enterprise boundaries, we assume, unlike traditional p2p based system where a node leaves the network very often (e.g., when an user went to sleep after shutdown his computer), a broker of MQM or PBM leaves the network only occasionally when it is taken down for regular maintenance or upgrade or when a (very) rare failure occurs. The analysis of this rare case does not have much value. Rather the analysis of the ideal case (without failure) which contributes all most all the time is sufficient to know who perform better: PBM or MQM.

Let us assume that the size of a message and its acknowledgement are l_m bits and l_a bits respectively including the protocol headers. We assume that all the broker to broker links has a constant data rate p bps. We ignore the delay of a message caused by the LAN within a site. Let us also assume that the average propagation delay caused by the distance between two brokers is t_d sec, the average storing (disk write access) delay is t_s sec and the average number of hop counts for PBM and MQM are h_p and h_m respectively. We ignore the storing delay if a message is stored in main memory. We also ignore various optimizations that can be used (for both PBM and MQMs) to improve performance.

3.1. Messaging Delay in PBM

Receiving Application is Online

In PBM, if the destination application is online a message is delivered directly to it. Therefore the average delivery delay

t_t = *time to transfer to the source broker + time to transfer from source to destination broker + time to transfer from the destination broker to the destination application*

$= (h_p+2)\ (2l_m/p + t_d)$

The time to transfer the acknowledgement from the destination broker to the source broker is

$t_a = (h_p+2)\ (2l_a/p + t_d)$

Therefore, the round trip time (RTT) is

$t_{rt} = t_t + t_a = (h_p+2)\ (2/p(l_m + l_a) + 2t_d)$

This delivery time t_t and the RTT t_{rt} is for all type of messages assuming that the destination application is online.

Receiving Application is Offline

When the destination application is offline, in PBM, the destination broker replicates the message into closest *K-1* leaf set members, stores it in its own memory and sends an acknowledgement. Therefore in this case:

t_{rt} = *message transfer time from sending application to destination broker + replica transfer time + replica acknowledgement transfer time + message acknowledgement transfer time*

$= (1+h_p)\ (2l_m/p + t_d) + (2l_m/p + t_d) + (2l_a/p + t_d) + (1+h_p)\ (2l_a/p + t_d)$

$= (h_p+2)\ (2/p(l_m + l_a) + 2t_d)$

We assume here that replication to *K-1* node is done in parallel. We ignore a small additional delay caused by sending *K-1* replica sequentially through a single channel. As we see that it is same as RTT for message that is directly delivered to the receiving application. However, this expression not valid for express messages because the express messages are not replicated. Therefore, for express messages:

$t_{rt} = (1+h_p)\ (2l_m/p + t_d) + (1+h_p)\ (2l_a/p + t_d)$

$= (h_p+1)\ (2/p(l_m + l_a) + 2t_d)$

3.2. Messaging Delay in MQMs

In MQMs, delay between application and the broker is negligible as they reside in the same site. But in case of transactional and recoverable messages additional delay is added due to storing of a message in each broker from source to destination application. When the acknowledgement is sent back through the reverse path, another disk access is needed in each broker. Therefore, for MQMs

$t_t = h_m (2l_m/p + t_d) + t_s(h_m+1)$

$t_a = h_m (2l_a/p + t_d) + t_s(h_m+1)$

Therefore, the round trip time is

$t_{rt} = t_t + t_a = 2t_s(h_m+1)+ h_m (2/p(l_m + l_a) + 2t_d)$

We assume here that message storing time and access time during acknowledgement is same. In fact the difference is negligible for small message size.

t_t and t_{rt} are for recoverable messages. The transactional messages may require more time as it follows a complex protocol. However, in MQMs express message requires less time as it is not stored in persistent store. For express messages:

$t_t = h_m (2l_m/p + t_d)$

$t_a = h_m (2l_a/p + t_d)$

Therefore, the round trip time is

$t_{rt} = h_m (2/p(l_m + l_a) + 2t_d)$

3.3. Traffic Generated for a Message in PBM

We would like to calculate, for a single message transfer from the sending application to the receiving application, how much inter-site traffic (the message, replicas and the acknowledgements) is generated in PBM. We ignore the traffic that is limited within a site.

Any communication between brokers is treated as an inter-site traffic. Unlike MQMs, in PBM communication between application and broker is assumed to be inter-site because in all most all the cases an application does not reside in the same site of its responsible broker. Therefore, the total inter-site traffic if the destination is online:

$l_{is} = l_m(h_p+2) + l_a(h_p+2) = (h_p+2)(l_m+ l_a)$

Total number of messages $n_{is} = 2(h_p+2)$. This is valid for all type of reliable messages. For unreliable messages as acknowledgement is not necessary, the inter-site traffic and the number of messages will be $l_m(h_p+2)$ and (h_p+2) respectively.

However, if receiving application is offline, extra messages are necessary to replicate, to send an acknowledgement from destination broker to destination sending application, to send a second acknowledgement when the message is delivered after the destination application comes online and to delete the replica. Therefore in such cases for recoverable (reliable) and transactional messages:

$l_{is} = l_m(h_p+1) + (K-1)(l_m + l_a) + l_a(h_p+1) + l_m + l_a (h_p+2)+l_a(K-1)$

$= l_m(h_p+K+1) + l_a(2h_p+2K+1)$

We assume here that the replica deletion message size is same as acknowledgement. Total number of messages $n_{is} = (3h_p+3K+2)$. For reliable express messages

$l_{is} = l_m(h_p+1) + l_a(h_p+1) + l_m + l_a (h_p+2)$

$= l_m(h_p+2) + l_a(2h_p+3)$

Total number of messages $n_{is} = (3h_p+5)$. Therefore, for unreliable express message $l_{is} = l_m(h_p+2)$ and $n_{is} = (h_p+2)$. But for unreliable recoverable messages they are $l_m(h_p+K+1) + l_a(2K-2)$ and h_p+3K-1 respectively.

3.4. Traffic Generated for a Message in MQMs

For MQMs, the amount of inter-site traffic is almost same for all type of reliable messages. In MQMs as an application and its responsible broker resides in the same site, inter-site traffic is reduced. However, every broker, when receives a message, must send an acknowledgement first. Additionally, as all the messages are stored in the destination broker, every (reliable) message is acknowledged more two times, one when stored by the destination broker, another when delivered to the receiving application. Compared with PBM, this increases the traffic significantly when the receiving application is online. Therefore, for MQMs:

$l_{is} = l_m h_m + l_a h_m + l_a h_m + l_a h_m$
$= l_m h_m + 3 l_a h_m$

Total number of messages $n_{is} = 4h_m$. For unreliable messages no final acknowledgements are necessary. Therefore, the amount of traffic will be reduced much. Therefore, for unreliable express and recoverable messages $l_{is} = h_m(l_m + l_a)$ and $n_{is} = 2h_m$

3.5. Summary

As we see one of the main factors that affect the delivery delay and RTT is the propagation delay t_d. Therefore, if the routing algorithm dispatches the messages through short paths, messaging performance should be improved. In case of MSMQ, as it uses a least-cost path from source broker to the destination broker, the average t_d will be very low. PBM can not choose a least-cost path but can chose a low cost path as it take routing decision based on a proximity metric. Moreover, link between an application and its responsible DIS is not optimal because it is not chosen based on proximity metric rather on a hash value. Therefore, in PBM the path of a message is much longer than that in MSMQ. However, PBM optimizes it by not storing messages in the persistent storage through the path way. In PBM a message is replicated by the destination broker only if the receiving application is not online. Whereas in MQMs, every reliable message is persisted in every broker from the source to the destination application.

Other than propagation delay t_d and storing delay t_s, another factor that affects both performance and message overhead is the hop count. Our Pastry based approach performs better than MQMs in this regard because hop count increases not linearly but logarithmically in PBM. As we are using replication based approach, the value of K, that is, how many brokers will contain the replica of a single broker is also a factor that affects the performance and message traffic.

4. EVALUATION

We have shown that our middleware can tolerate failure of a broker. If a broker fails, there always has a broker who can take the responsibility provided that at least one broker is alive in the system. In this section we show that despite providing such facilities, our middleware claims lower deployment cost and generates less traffic. The messaging performance is also reasonable.

To know how PBM performs we design and run some experiments using OverSim[13] in Omnetpp[12] simulation environment. Our simulation scenario is as follows: we consider that a large enterprise deploys a messaging system in its many sites located over a large geographical area. Each site has a broker and a number of client applications connected in a LAN. We obtain the *keys* by applying a *sha1* hash function on the names of the applications and the *nodeIds* by applying the same hash function on the IP address of the brokers. An application sends a message to another application via the responsible broker whose *nodeId* is numerically closest to the application's *key*. To understand the effect correctly we balance the load, i.e., we set the simulation in such a way that each broker is responsible for equal number of applications and each application sends one message for each type to every application including itself.

We set various parameters as follows: the Pastry configuration parameter *b* called *bits per digit* is 4, both *number of leaves* and *number of neighbors* is set to 16. We choose disk write access time randomly from 5 ms to 10 ms range [19]. To simulate the delay between brokers, at the starting of the simulation, we fix the delay from each broker to every other broker randomly. The random value ranges from 1 ms to *maxDistance* ms where maxDistance is the *one-way* propagation delay (in ms) between the farthest brokers. We put these values in a distance matrix. We do not use a two dimensional plane to simulate the distance because, in real situations, triangular inequality does not hold. The propagation delay does not include the transmission and reception delay caused by the channel bandwidth. We assume that all channels between brokers have a bandwidth/data rate of 3.152Mbps.

We compare PBM with MSMQ to understand how the system would perform if MSMQ were deployed in case of PBM. We use the same distance matrix used in PBM. MSMQ uses a least-cost path algorithm to route a message from one broker to another. We consider propagation delay as the cost. We do not include storing delay in the cost, because express messages are not stored. As MSMQ is configured and maintained manually we assume that from every broker there are direct links to 5 other brokers. We assume that the administrator is careful enough (although in real it is very difficult) to choose the least-cost links as the direct links. We choose smallest 5 values from each row of the distance matrix for 5 direct links. Based on these direct links we run Dijkstra's algorithm to find the least-cost path from each broker to every other broker. These least-cost paths are used for messaging between applications in our MSMQ simulator.

4.1. Scalability

To know how PBM perform as the broker network grows in size, we fix the maximum one way propagation delay (indicates the length of the geographic area) to 15 ms and vary the number of brokers from 64 to 1024. Then we plot the one way average delivery delay in Fig.6(a) and round trip time in Fig.6(b) for different types of messages in log scale (with base $2^b = 16$). If the receiving application is offline, we measured the delay between the sending application and the destination broker.

As we see from both figure 6(a) and 6(b) that the growth of one way delivery delay and RTT in MSMQ is more rapid than that in PBM. This indicates that as the number of brokers grows PBM performs better than that of MSMQ. Also for maximum propagation delay of 15 ms and for all type of messages except for express messages, PBM perform much better than MSMQ especially for higher number of brokers. But as we will see in a subsequent experiment that in very long geographic areas our middleware does not perform well.

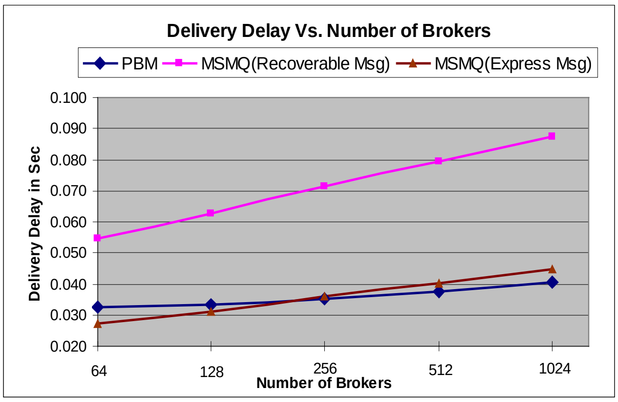

Fig. 6: (a) Delivery delay Vs. Number of brokers

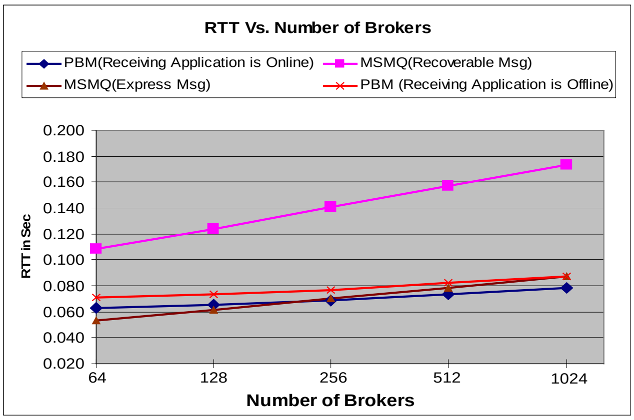

Fig. 6: (b) RTT Vs. Number of Brokers

4.2. Traffic Generation

In the same experiment for measuring scalability, we measure how many inter-site messages (original messages, replicas and the acknowledgements) are generated for transferring a single message from one broker to another broker. This experiment gives an idea how much traffic is generated in PBM compared to MSMQ. We assume that, in PBM, the value of $K = 3$, *i.e,* if the receiving application is offline, the destination broker replicates into $K-1 = 2$ closest (numerically) brokers and store it in itself. Our assumption $K = 3$ is reasonable because in order for a long term service loss of the applications in a site all 3 brokers must fail within a very short time. The probability of such occurrence is extremely low as those 3 brokers are dispersed geographically. In PBM network with $K=3$ is more resilience than the network in MSMQ with three cluster members in each site because PBM is not affected by a disaster while MSMQ does. We plot the number of messages generated in reliable and unreliable messaging in Fig.7(a) and Fig.7(b) respectively.

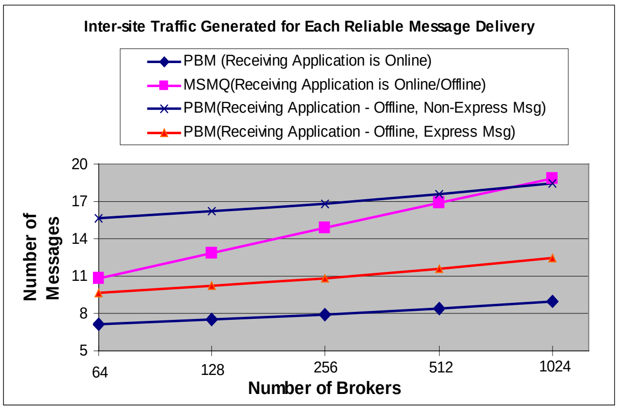

Fig. 7: (a) Inter-site traffic generated for each reliable message delivery Vs. number of brokers

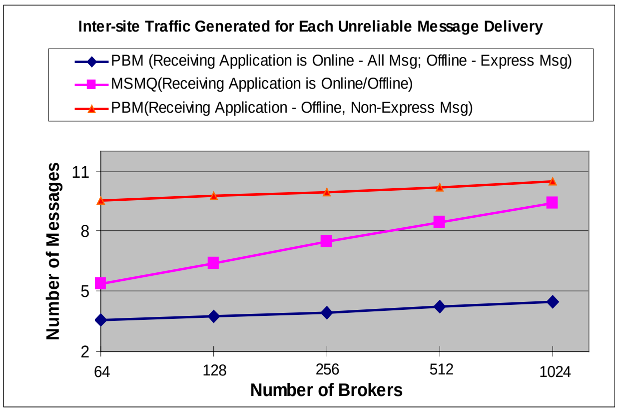

Figure 7: (b) Inter-site traffic generated for each unreliable message delivery Vs. number of brokers

As we see from the figures that in MSMQ the number of generated messages grows very rapidly than that in PBM. This is another proof that PBM is more scalable. In MSMQ the number of generated messages is same if the receiving application is offline or online because all the messages are stored. In contrast, in PBM if the receiving application is online it requires much less number of messages as the message is not replicated. However, if it is offline, a slightly higher number of messages are required as the message is replicated. Even though, if the number of brokers is above 1000, MSMQ performs worse than PBM. The usual case should be that for most of the messages the receiving application is online. Therefore, on average PBM generates much smaller number of messages than MSMQ.

4.3. Effect of Broker to Broker Distance

As we have stated that messaging performance is affected by the one way propagation delay between two communicating brokers. In this experiment we compare the messaging delays with that in MSMQ for various propagation delays. We vary the maximum propagation delays from 10 ms to 60 ms and plot the messaging delays in Fig.8(a) and Fig.8(b). The maximum propagation delay indicates how large the geographic area is where the messaging system is deployed.

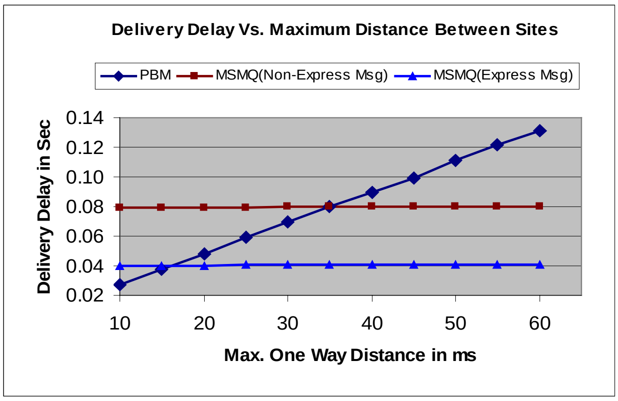

Figure 8: (a) One-way delivery delay Vs. Size of the geographic area in term of propagation delay between the farthest brokers

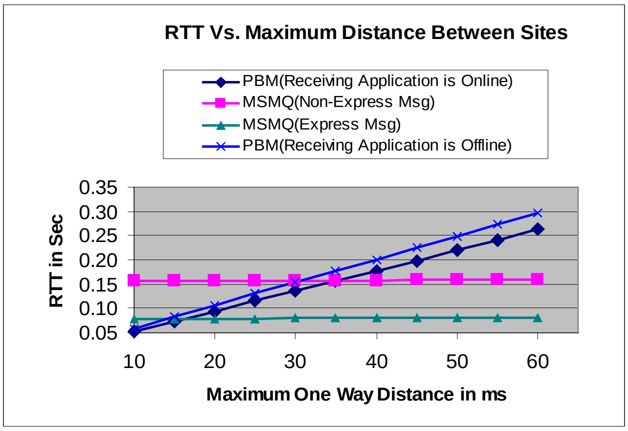

Figure 8 (b): RTT Vs. Size of the geographic area in term of propagation delay between the farthest brokers

As we see from the figures, in PBM the one way delivery delay and RTT varies rapidly as the maximum distance increases. However, in MSMQ they are almost constant because the messages are always getting a least-cost path which is slightly affected if the average propagation delay increases. As we see in MSMQ express messages have very good performance which is comparable with PBM only if deployed within a limited geographic area. However, in a messaging system transactional and recoverable messages should dominate. For these types of messages our middleware performs very well within a geographic area bounded by the propagation delay about 35 seconds. Over this value, MSMQ perform well. According to our measurement using *ping* command (in peak hours), propagation delay of 35 sec (hence RTT is about 70 sec) covers a medium sized country boundary e.g. Japan (far ends of). Therefore, within such a country boundary PBM performs better than MSMQ.

4.4. Administrative Overhead

As we see that in MSMQ, in order to populate the routing table by the routing algorithm, a number of input tables are needed, e.g., *SiteRecordTable*, *RoutingLinkRecordTable*, *MachineRecordTable*. Such tables need to be maintained manually by the site administrator [7]. This might be easy if the messaging system is deployed in a limited number of sites. As the number of sites grows, maintaining such tables manually becomes very difficult and time consuming. Our proposed PBM does not suffer from this problem. Here nothing needs to be maintained manually. The routing tables, neighbor sets, leaf sets, queues, applications' connection information all are updated automatically as the broker or application leaves or arrives. Although this requires some extra traffic overhead due to periodic communication among brokers, MSMQ also is not free from such overhead. For example, it needs active directory service which generates some extra traffic to maintain the service.

TABLE 2: PER SITE COST COMPARISONS BETWEEN PBM AND MQM

Type of Deployment	Hardware	Quantity	Typical Price (USD)	Total Price
PBM	Application Server	1	2000	2000
MSMQ with Clustering	Application Server	2	2000	4000
	Database Server	2	3500	7000
MSMQ without Clustering	Application Server	1	2000	2000

4.5. Deployment Cost

Unlike MQM, PBM does not require that every site must install a broker. Therefore, PBM can be deployed with very little cost. Even if we consider that every site should have a broker, deployment cost is not higher compared to MQM. If a cluster based deployment is used for MQM, each site must have at least two application servers and two database servers assuming that the popular master/slave clustering is used. If a non-cluster based deployment is used, each site requires one server resulting in equal cost of PBM. Table 2 shows per site cost comparison between PBM and MQM assuming that as an application server Dell PowerEdge 2970 and as a database server Dell PowerEdge 2950 III is used. As we see for a cluster based deployment MQM requires more than five times investment compared to PBM excluding the disaster recovery management cost. However, a non-cluster based deployment of MQM costs same as PBM. However, such deployment can not provide resilience to broker failures.

4.6. Discussion

The scalability experiments shows that as the number of broker increases, messaging delays grow slowly in PBM compared to that in MSMQ. However, in the last experiment we see that PBM perform better only in a country boundary. These two results are not contradictory. They mean that PBM performs well if more number of brokers are installed within that boundary. For example, if there are 1000 sales centers of an enterprise over a country, they can deploy a PBM for better services compared to MSMQ. Outside this boundary although PBM can not provide better performance but what it can provide is lesser traffic generation, lower deployment cost, automatic failover, minimum administrative overhead.

5. RELATED WORK

We have not found any work which considers the issues directly related to the currently available MQMs. JMS[20], AMQP[21] tries to standardize communication between applications and brokers. But they do not define how the message should be routed (which is our main concern) between brokers distributed in a network. However, we have found several works related to messaging systems built on p2p networks. P2P based systems are used mainly to provide persistent shared storage services to the clients, e.g., CFS[14], PAST[15] etc. In such systems, unlike our use, a file is stored/uploaded once but accessed many times. Therefore, file searching, efficient use of storage are some of the main issues of shared storage which are not issues for our system where a message will be deleted from the queue once it is delivered. P2P network is also used as a middleware for multicast/anycast or publish-subscribe based systems, e.g., Hermes[16], REM[17], SCRIBE[18], REBECA[]. We use Pastry as a point to point message queuing middleware (not as publish-subscribe). Our proposed middleware needs to solve issues related to reliable, in-order and exactly once delivery semantics. Some instant messaging systems, e.g., DIMA[8], which is partially related to our work, have been built on Pastry but they have not considered those issues. Another related work is POST[9], a general purpose messaging system based on Pastry. POST uses store-and-forward architecture and can provide multi-cast communication. However, POST has some limitations. It does not consider in-order delivery issue as it is not a message queuing middleware. Each of the messages is stored in persistent storage and replicated to a number of brokers compared to of only those messages that can not be delivered in our middleware. As the cost of storing and replicating in a network is very high, our system should be much faster than POST. Besides, sending a message must be followed by a notification message consuming more bandwidth. If the destination application of a notification is not alive, it adopts a costly approach to deliver the notification.

6. CONCLUSION

We have proposed a novel approach to design MQMs based on Pastry peer to peer protocol. Our middleware eliminates a number of problems of traditional MQMs. It failovers automatically to another broker located in a different site if the current broker fails, it eliminates the administrative overhead necessary to maintain the broker network. Experimental evaluation shows that such services can be obtained with reduced traffic overhead and that our middleware is especially appropriate for a network of large number of brokers deployed in a country boundary.

However, we have to consider some more issues. As the states are not necessarily to be persisted, if a broker is crashed and losses its memory content or if it is restarted, it must get necessary state information and replicas from its replica set members. Besides, as the network is self-managed, in some cases of link failures, it can create a network partition. We have not considered it yet although existing methods are available to guard against such partitions. Also, we have to consider how a point to multi-point communication can be provided based on our point to point service.

7. REFERENCES

[1] An Introduction to Messaging and Queuing. Document Number: GC33-0805-01, IBM, June 19995. ftp://ftp.software.ibm.com/software/mqseries/pdf/horaa101.pdf

[2] MSMQ, Microsoft Developer Network Library. http://msdn.microsoft.com/en-us/library/ms878320.aspx.

[3] Group Discussions in the group microsoft.public.msmq.networking, http://www.microsoft.com/communities/newsgroups/list/en-us/default.aspx?dg=microsoft.public.msmq.networking

[4] High-availability cluster. (2008, June 24). In Wikipedia, The Free Encyclopedia. Retrieved 05:57, June 30, 2008, from http://en.wikipedia.org/w/index.php?title=High-availability_cluster&oldid=221338263.

[5] Active MQ. http://activemq.apache.org/.

[6] Open Message Queue. http://www.mq.dev.java.net/.

[7] Message Queuing (MSMQ): Binary Reliable Message Routing Algorithm, http://msdn.microsoft.com/en-us/library/cc235039.aspx

[8] H. Lundgren, R. Gold, E. Nordström, M. Wiggberg. A Distributed Instant Messaging Architecture based on the Pastry Peer-To-Peer Routing Substrate, In Proc. of Swedish National Computer Networking Workshop, Stockholm, Sept. 2003.

[9] A. Mislove, A. Post, C. Reis, P. Willmann, P. Druschel, D. Wallach, X. Bonnaire, P. Sens, and J. Busca. POST: a secure, resilient, cooperative messaging system, Proc. of the 9th conference on Hot Topics in Operating Systems, pp. 11-11, Hawaii, May 18-21, 2003.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest. Introduction to Algorithms , The MIT Press, 1990.

[11] A. Rowstron, P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In Proc. of IFIP/ACM Middleware, pp. 329–350, Nov. 2001.

[12] OMNet++. Discrete Event Simulation System. http://www.omnetpp.org/.

[13] OverSim: A Flexible Overlay Network Simulation Framework. http://www.oversim.org/.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In Proc. of 18th ACM Symposium on Operating Systems Principles, pp. 202–215, Oct. 2001.

[15] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. HotOS VIII, Schoss Elmau, Germany, May 2001.

[16] P. R. Pietzuch and J. M. Bacon. Hermes: A distributed Event-Based Middleware Architecture. In the Proc. of the 1st International Workshop on Distributed Event-Based Systems, 2002.

[17] M. R. Selim, Y. Goto, and J. Cheng. A Replication Oriented Approach to Event Based Middleware Over Structured Peer to Peer Networks, Proceedings of the 5th International Workshop on Middleware for Pervasive and Ad-Hoc Computing, A Workshop of ACM/IFIP/USENIX 8th International Middleware Conference (MPAC 2007 of Middleware 2007), pp. 61-66, Newport Beach, USA, ACM Press, Nov. 2007.

[18] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralised application-level multicast infrastructure. In IEEE JSAC, 2002.

[19] Average Random Access Time (Write), Drive Performance Database. http://www.storagereview.com/.

[20] Java Message Service (JMS), Sun Java Developer Network, http://java.sun.com/products/jms/.

[21] Advanced Message Queuing Protocol, http://amqp.org/.

[22] REBECA, http://kbs.cs.tu-berlin.de/projects/rebeca_engl.htm/.

Article received: 2008-07-12