# A Highly Available Message Queue

Ashima Munjal

munjal@jhu.edu

January 22, 2003

Advisor: Dr. Yair Amir

yairamir@cs.jhu.edu

Submitted as partial fulfillment of the requirements for the degree of
Master of Science in Engineering from The Johns Hopkins University

## 1 Introduction

The ability to exchange messages via the Internet has become a basic necessity in our day to day lives. Just as individuals exchange E-mails to keep up with their fellows, several applications available today use various messaging systems to keep their information current and consistent. This document proposes building a highly messaging system that enables multiple clients to publish and subscribe messages, replicates messages at available nodes and does not have a single point of failure. The distributed system would provide the flexibility of initiating and retrieving messages at any given node.

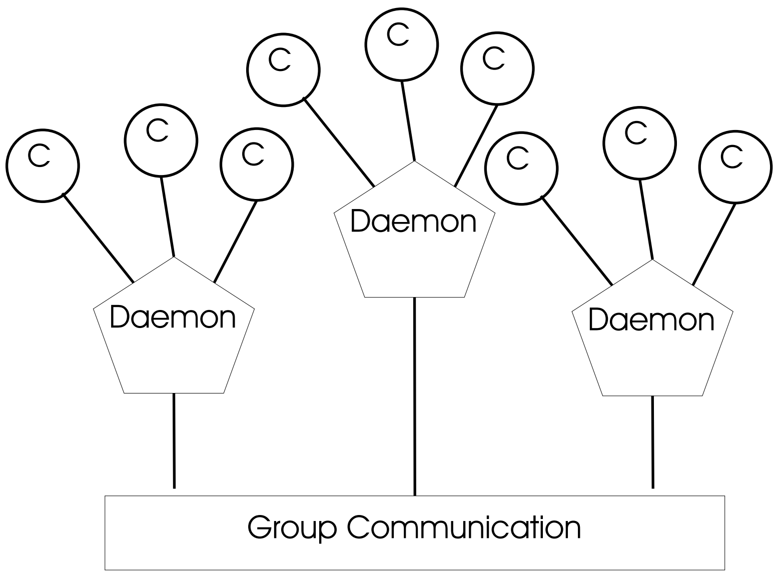
The proposed system strives to achieve the same view of available messages on all configured nodes, and cope with system crashes and network partitions. It accepts messages from clients, secures them on stable storage, passes them around to other nodes and distributes them to connected clients. Each node passes state information and data messages to other nodes as soon as possible (under the given network conditions), but it uses a fairly lazy scheme for message annulment information as it assumes that disk space is not a scarce resource.

The system provides first in first out (FIFO) service to all clients publishing messages. Clients may publish messages to any topic[1], but to receive messages they must subscribe to the topic. A subscription may be temporary or persistent. In case of a temporary subscription the system makes the best effort to deliver all messages while the client is connected to the system, but only provides at most once guarantee. In case of a persistent subscription, the system holds a copy of each message until it is delivered to the client. If the client connects to the same physical server, the system can provide exactly once guarantee. However, if the client moves from one system to another, then it must tell the server what messages it has received for the exactly once guarantee.

[1]A topic is an administrative object that relays messages to the set of clients that register subscriptions

## 2 System Overview

Figure 1: Basic Architecture



The system uses multiple nodes to build a highly available message queue while trying to present the notion of a single reliable entity to its clients. There are multiple daemons that run on the configured nodes, that allow clients to connect to them. From the clients perspective, connecting to any one of these daemons is mostly[2] the same as connecting to any other. These daemons communicate with each other using a group communication system, namely the Spread toolkit which is described in more detail in Section 4.1.

To efficiently distribute messages, we create topics that act as message brokers between different clients. This allows for a division of load, for different topics may have distinctive lag and storage requirements. Suppose there is only one publisher and one subscriber, and the subscriber keeps pace with the publisher. The number of messages stored for

[2]If the client changes the daemon it is communicating with, it must provide an uptodate copy of its message headers to preserve exactly once delivery. Otherwise, some messages may be redelivered.

this topic and the overhead involving the calculations the servers have to make is rather low. On the other hand, if there are many publishers and subscribers for a topic, all messages will be kept around for the slowest client. Therefore, slower clients will only delay things in the topics they subscriber to, and not all messages.

To create and delete topics, we have a more privileges topic called "directory". This is a default on all nodes. An administrative interface is used to make changes on this topic, and it is replicated on all nodes. While the updates to this topic may only be made by an administrator, all the clients can access it to get information about the current set of topics. To avoid confusion on the state of directory information, full access is restricted to the administrator. If the same topic is created at two nodes while they are unable to communicate with each other, the data is simply merged. However, when it comes to deleting topic, thing are no so simple [3].

All messages published to a topic must be delivered to clients in an order that preserves FIFO guarantees. This means that messages from any publisher to the same topic must be delivered in the order they where sent to all the subscribers. However, messages from different publishers for the same topic may interleave in any random fashion. Also, this guarantee makes no statement about the messages published to different topics by the same subscriber. To order messages is rather simple, as they are sorted by the sequence number created for them by the daemon that publishes them. But, deciding when a subscriber joined the system and what is the first message it receives from each publisher is a bit more cumbersome.

In order to publish a message to any given topic, a client must connect to a daemon and send the message to that topic. When the daemon receives the message from the client, it creates a sequence number for that message. Thus, every data message has a unique

[3]More details on this in Section 3

combination of sequence number and origin id. When a client subscribes to a topic, each daemon must acknowledge its subscription. The new subscriber receives the data messages incepted at a daemon only after the daemon sent the subscription acknowledgment messages to its peers. This way, when a message originates at a daemon, the daemon knows the exact set of recipients for that message i.e. the message can be deleted after this set of clients receives it. In short, when a new subscription is created, a unique starting point for messages originating at the various daemons is determined and FIFO guarantees are thereby provided.

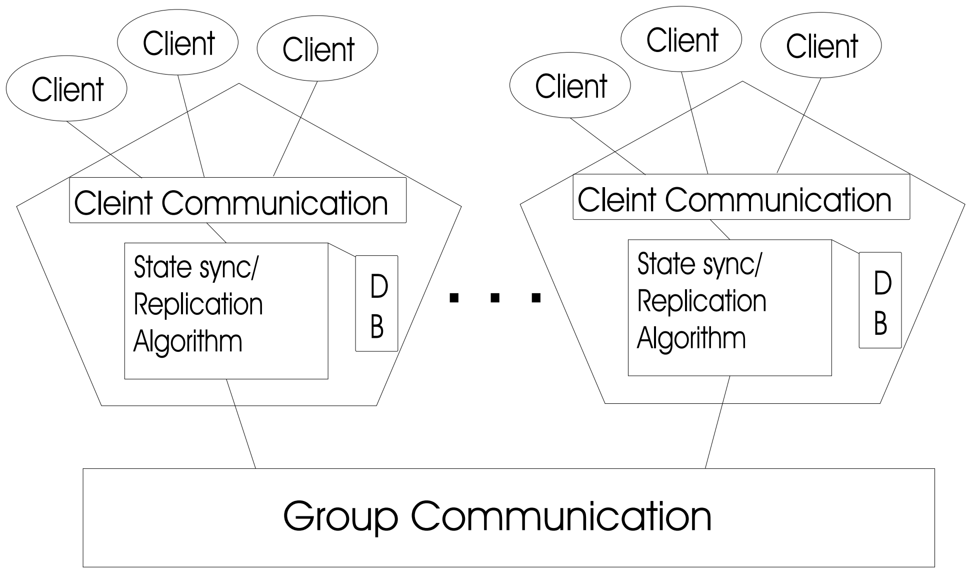
All the nodes that run daemons that provide a highly available message queue are pre-configured. Therefore, the maximum number of nodes are known and all the daemons communicate using the group communication system which is also pre-configured. Thus, all calculations can be made deterministically. They also assume that ample amount of disk space is available so that each topic can be hosted everywhere. However, it is possible to host a topic at selected nodes only with certain variations to the algorithm. It is also assumed that a message gets from one daemon to another (and thus to clients connected to different daemons) if there is an eventual path between the two daemons. In other words, two daemons need not be directly connected for an exchange of messages between them. As long as pairs of processes are directly connected at successive periods in time, the messages are passed around. The algorithm that achieves this is described in Section 3.

## 2.1 Architecture

The main components of this system, as presented in Figure 2 are:

- A client communication interface

Figure 2: System Architecture



- The replication and state synchronization algorithm
- A database manager
- A group communication system

The client communication interface receives clients requests asynchronously and passes them on to the replication and synchronization algorithm. The main algorithm processes the request, and uses the client communication interface to send messages back to the client. It also uses the database manager to store messages and topic and client state information on stable storage like a hard drive. It uses the group communication system for membership information, and as a message bus for exchanging information with other daemons. The algorithm presumes that the group communication system presents an accurate view of the network in the membership information it provides and that it delivers the messages in FIFO order to other daemons.

The replication and state synchronization algorithm maintains information about the different topics, that includes the last message incepted for each topic at every node, the clients subscriptions to the topic and the last message received by each client. This information is passed around to other daemons when the group communication system reports a membership change. Based on that, the daemons decide what messages need to be re-sent to update other daemons, and this way a messages are received as long as there is an eventual path between two daemons. Although not required for theoretical correctness, state information is also periodically piggy backed on data messages to keep daemons up to date.

The client state information is exchanged less often, as it is primarily used for garbage collecting messages that have been consumed by all clients. Another consequence of this is that if clients change the node they are connected to, then the new daemon may not know which was the most recent message delivered to that client. Therefore, for exactly once delivery, the client may have to provide the daemon with some information.

On a practical note, each of the nodes running this system must run the Spread Toolkit as a process, and the persistent-messaging daemon as another process. They must allocate local disk space for storing messages and other meta information. Since neither the Spread Toolkit or persistent-messaging daemon support dynamic allocation of nodes, the configuration files must be complete from the very beginning.

## 3 The Replication and State Synchronization Algorithm

The persistent messaging daemon runs a Replication and State Synchronization algorithm that gives higher priority to the topic “directory”, for this topic stores meta information regarding the other, more dynamic topics in the system. After each membership change reported by Spread, first the directory information is synchronized to ensure that all the replicas are up to date and then a slightly modified algorithm is run for the rest of the topics. The main difference between directory messages and other messages is that directory messages are deleted once all the daemons specified by the configuration apply it to their state. However, messages related to other topics are deleted only when all the clients that have subscribed to them either receive those messages or unsubscribe.

As mentioned earlier in Section 2, messages produced or incepted at one daemon will be delivered to another daemon as long as there is an eventual path between the two daemons. The delivery of pending messages also depends on the time duration of the network connection between daemons. As stated earlier, directory messages are delivered before other messages. An eventual path may be a direct network connection between two daemons, or a connection between different sets of processors at successive periods of time e.g. if daemons $a$, $b$ and $d$ are connected to each other during time interval $t_1$, all messages produced by $a$ are shared with $b$ and $d$. Later on, if $c$ has a direct connection with $d$ during time interval $t2$, it will receive all the messages $d$ had in $t_1$, as long is $t_2$ is long enough to transmit those messages on the network. Thus, through $d$ there exists an eventual path to $c$ for the messages produced at $a$ and $b$ during time $t_1$. Of course, there is direct path between $c$ and $d$.

A data messages is produced or incepted in the system as a consequence of a client message. Therefore, the client interface, as described in Section 3.1 must be considered before the algorithm run by the daemons can be understood. The algorithm run for the directory information is described in Section 3.2. The difference between the algorithm run for the directory topic and the rest of the topics are described in Section 3.3.

## 3.1 The Client Interface

An administrative client must have the ability to

- get a list of topics
- create a new topic
- delete an existing topic

A client must have the ability to

- get a list of current topics
- subscribe to any given topic
- send a message to any given topic
- unsubcribe to a given topic
- change subscriptions
- receive messages

A client may send messages to any topic, whether or not it has subscribed to that topic i.e. topics are open to receiving messages from anybody. However, a client can only subscribe to one topic per session, and each session can only have one active connection. A client can subscribe to multiple topics by openening multiple simultaneous sessions[4]. The client API transparently converts a change on subscription to a unsubscribe request and a subscribe request. The API allows the clients to receive messages by a blocking receive call, or by starting a message listening thread.

[4]this is required for JMS compatibility and is discussed in more detail in Section 4.2

## 3.2 The Directory

Messages that alter directory information are received by a daemon from a client, and then distributed to all other daemons. A *directory message* may be destroyed once all the daemons have applied it to their directory state database. To achieve this, as an administrative client sends a message to a daemon, it creates a corresponding daemon level message called *directory message* that includes the *server's id*, a serial *message index* generated by the server, and a *lamport time stamp*[2]. After the *directory message* is created, it is first applied to the daemon's local directory database. Storing the newly created *directory message* in the database of messages and changing the directory state is done in one single transaction. The newly created message is sent to all other daemons using the group communication service. This message is piggy backed with the lowest lamport time stamp that the server knows off, from any damon.

The combination of *server id* and *message index* are used to ensure that the daemons are current with *directory messages*, and that messages are appropritately retransmitted after a membership change. The *lamport time stamp* helps effectively detele *directory messages.*

Each daemon stores a vector of last directory message index that it received from other daemon. This vectory is updated as the server receives *directory messages.* When the group communication system reports a membershrip change, a *state message* containing the directory message vector is sent out. After all the state messages from the currently connected members are received, each server calculates what messages need to be resent. If there is a cascading memberhip change i.e. another memberhsip change while the state messages are being exchanged, new state messages are sent. Each daemon calculates the lowest and highest *directory message* sent by all the daemons in the system. Then they retransmit the messages. Thus, after a membership change and one round of state messages, the server know what messages need to be retransmitted. And if there is enough connected time, there messages are retransmitted.

As various daemons receive *directory messages*, they record the highest lamport time stamp that they have heard from that daemon in a vector, and then the update the lowest lamport time stamp (min lts) that they have heard form any daemon. Each server also advertises its min lts and then the least of the min lts is calculated. This is the all received upto lamport time stamp (aru lts). All messages marked with a lamport time lower than the aru lts can be safely deleted.

The min lts vector represents what each daemon knows about the other daemons in the system. And the aru lts, which is the lower bound for min lts, represents global knowledge. This works in case a server has crashed, as messages that are time stamped after it will have an lts above the crashed server's min lts. Therefore, new messsages created after the aru lts was altered, are kept around until the crashed server recovers. This also works in case of a partition, as each side will not be able to update the min lts of the daemons they are partitioned from.

Here is a brief description of the data structures related to the directory.

**server_id** is a unique indentifier held by each server

**current_lts** is the current lamport time stamp.

When a new mesage is received, current_lts = max(message.lts, current_lts)

When a new directory message is incepted, it is marked with

lts = current_lts = current_lts + 1

**directory_index** is a vector that stores the last index of messages created by each daemon.

**directory_lts** is a vector that stores the maximum lts that has been created by each daemon.

**directory_aru** is the all received upto lts that is advertised by daemons. For any given daemon, its aru = min(directory_lts vector)

**topic_info** is the list of topics served by the highly available message queue.

## 3.3 Dynamic Topics

The dynamic topics enable applications to exchange messages among themselves. Once created by an administrative user in directory, the dymanic topic runs an algorithm that is very similar to the directory, but unlike the directory, the dymanic topics cannot delete messages when all the servers have received them. The messages can only be deleted when they have been delivered to all persistent subscriptions.

As described in Section 2, the client messages and client update messages share the same message queue to achieve FIFO symantics. Client messages can be deleted only after all the clients have received them, and client update messages can only be deleted after all the daemons have received them. Therefore, we maintain two vectors related to deletion. One related to the messages that have been delivered to clients (client messages) and the other related to messages that have been received by all the daemons (client messages + client update messages). The mesages that have been received by all the deaemons are computed like directory messages. The other vector is updated each time a client acknowledges it has received certain messages. If there are messages that have been delivered to all the subscribers, but not received by all the daemon, they are marked deteleted and the message body is shortened. It is only after a messages has been deliverd all the daemons and subscribers, it can be deleted completely. Partial deletion is prefered because a client messages can be quite space consuming.

## 4 Practical Considerations

A message queue must be highly responsive to be highly available. The biggest practical challenge in implementing this system is to avoid huge lag in client server communication. For example, when daemons merge, there may be a lot of messages to be exchanged between them. The inter-daemon communication will tend to dominate the usage of bandwidth. A happy median must be achieved in sending messages among servers and responding to clients.

In this system, client messages are put on hold while state messages are exchanged. Then the servers calculate what messages need to be resent, they start receiving client messages again. The messages should be resent at a rate that does not overwhelm the client messages completely and the service remains responsive. However, if there are membership changes are a very fast pace, the system will become non-responsive to clients. Therefore, network stability affects the availability of this message queue.

The groups communication system, as described is Section 4.1, is important for its message thoughput affect the thoughput of this message queue and it correctness affects the correctness of this system.

The message queue also fulfils many components fo the Java Messaging Sevice, and those are described in Section 4.2

## 4.1 Group Communication – The Spread Toolkit

Spread[1] is a general-purpose group communication system for wide- and local-area networks. It provides reliable and ordered delivery of messages (FIFO, causal,agreed ordering) as well as Virtual Synchrony and Extended Virtual Synchrony membership services.

Spread uses a client-daemon architecture. Node crashes/recoveries and network partitions/remerges are detected by Spread at the daemon level; upon detecting such an event, the Spread daemons install the new daemon membership and inform their clients of the corre-sponding changes in the group membership that are introduced by the failure. Clients are also notified when changes in the group membership are triggered by a graceful leave or join of any client. The Spread toolkit is optimized to support the latter situation without triggering a full daemon membership reconfiguration, but rather in-forming only the participating group about the new group membership.

The Spread toolkit is publicly available and is being used by several organizations in both research and production settings. It supports cross-platform applicationsand has been ported to several Unix platforms as well as to Windows and Java environments.

## 4.2 Java Messaging Service

Sun Microsystems has defined an API for sending and receiving messages in the Java environments called Java Messaging Service (JMS)[[3]]. JMS supports both point to point message exchange and well publish subscribe message queues. JMS also has a defined relationship with other Java APIs susch as Java Database Connectivity Software (JDBC), Enterprise Java Beans (EJB), Java Transaction API (JTI), etc.

The message queue described in this document fulfills all the requirements of the publish subcribe system in JMS. The point to point system can be implemented within the same system. However, this messaging system does not support the relationship JMS API has with other Java APIs. For the rest of this section, I will decribe how the components mentioned in the ealier sections map with the JMS API.

The directory topic is a rudimentary form of the Java Naming and Directory Interface (JNDI). It is used a get information about topics. The topics, and subcriptions are of course the same. In JMS, persistent subscriptions are known as durable and non-persistent subscriptions are non-durable. Topic administration is done via the directory topic. JMS uses a topic session for creating topic publishers, topic subscribers and for unsubscribing from topics. Suppose we have a topic subscription session, all the messages for that topic must be delivered to this session. For this reason, we consider the client connection to the server as a topic session. This does limit the number of client the system can support to much less than the number of connections that the machiene can support, but it allows us to support multithreaded clients that want to recieve messages for serveral topics in a different thread. If a single client is publishing and subcribing to the same topic, it would need a publishing session and a subscribing session which translates into two different network connection. Besides reducing the complexitiy of the server implementation, this also allows us to have diffent error message handlers or exception listeners, as the are called in the Java world.

## 5 Related Work

There has been a lot of work in group communication that supports a publish subscribe system, but most of it does not provide persistence, though they are very efficient at best effort delivery. Most current systems that support a persistent publish subscribe system are based on a single server, and therefore do not support clustering.

One of the poineers in the group communication area is the Spread Toolkit, and that is why it is the underlying infrastructure for this system. Another important project is the ensamble project[4] developed at Cornell. It provides a library of protocols that can be used for building distributed systems, and it is built in a higly modular fashion.

Many commertial systems such as IBM MQ, SonicMQ, and open source systems such as JBoss offer message queues that provide exactly once message delivery symantics for messages and persistence. The open source projects do not support clustering; the commertical project don't offer any information available freely that suggest that they support clustering.

The Gryphon project[5] developed by IBM supports durable subscriptions in a publish syscribe system. It logs each message only once, and actively filters each message such that it only travels to brokers which have receivers. This is in much contrast to our approach of replicating everything everywhere. Ofcourse, its a trade off in storage space efficiency and avalablity of messages.

## 6 Conclusion

This document presents a software solution that provides a highly available message queue using a cluster of computers that may be geographically apart. The builds on the Spread Toolkit, a group communication, to provide persistence. This is also a limited Java Messaging Service Support. JMS compatibility allows the possiblity of JMS users to use this in existing systems without making many code changes.

## References

[1] Y. Amir and J. Stanton. The spread wide area group communicatoin system. Technical Report CNDS 98-4, Johns Hopkins University, Center for Networking and Distributed Systems, 1998.

[2] L. Lamport. Time, clock and ordering events in a distributed system. Comm. ACM, 21(7):558-565, July 1978

[3] Java Messaging Service - http://java.sun.com/products/jms/

[4] M. Hayden The Ensemble System. PhD thesis, Cornell University, 1998

[5] S.Bhola, Y. Zhao and J. Auerbach. Scalably Supporting Durable Subscriptions in a Publish/Subscribe System, IEEE DSN 2003.