# Distributed system project

----A Simple Queue Service

Team member

5090379001, Li Tingting

5090379073, Huang Tao

5090379085, Wang Jiajun

5090379107, Ban Yunmeng

5090379175, Li Wenhao

目录

[Distributed system project 1](#_Toc326332891)

[----A Simple Queue Service 1](#_Toc326332892)

[Team member 1](#_Toc326332893)

[Introduciton 3](#_Toc326332894)

[Design 3](#_Toc326332895)

[Event-driven library 5](#_Toc326332896)

[SQSClient 5](#_Toc326332897)

[SQSMaster 6](#_Toc326332898)

[SQSDataNode 7](#_Toc326332899)

[Testing 8](#_Toc326332900)

# Introduciton

We develop a simple queue service like Amazon SQS. It offers a reliable and scalable mechanism for distributed applications to communicate with each other. In particular, the applications now don't need to worry about possible message loss or the availability of queue service.  
The SQS allows multiple clients from different machines to send or receive messages from a queue. SQS guarantees that once a message is successfully sent by a client, the message will not be lost due to any one machine failures. To achieve this, SQS stores copies of clients' messages on multiple servers for redundancy and high availability. On rare occasions, one of the servers storing a copy of a message might be unavailable when clients receive or delete the message. If that occurs, the copy of the message will not be deleted on that unavailable server, and the client might get that message copy again when receiving messages. (Because of this, client applications must design to be idempotent, i.e., it must not be adversely affected if it processes the same message more than once).

SQS locks the message during processing, keeping other clients from processing the message simultaneously. SQS will provide some APIs to clients. Essentially, SQS provides reliable (at-least once delivery) message passing, but doesn't guarantee messages are received by any particular order. You can choose any programming languages to use, just make sure the code can run on the provided testbed.

# Design

We implement the project using c++ language and event-driven model programming with the help of libevent.

There are three main parts in our design :

SQSClient

SQSDataNode

SQSMaster

They represent the client end, datanode end, master end of the whole project. The client can read or write message in the queue. It first ask the master where the datanode is and the master give back a datanode address according to its record of available datanode. Then client will connect to datanode and do some operations, the datanode will respond. Then the node will send the change message to master for log. For each system, we use three datanodes to record the same data for the availability of the queue. Once one of the datanodes crushes, another one will replace its role.

**8**

**7**

**6**

**5**

**4**

**3**

**2**

**1**

DataNode

DataNode

DataNode

Client

Master

**7**

**8**

The details of each part are as follows.



Event-driven library

We use libevent to implement event-driven programming. It is an asynchronous event notification software library. The libevent API provides a mechanism to execute a [callback](http://en.wikipedia.org/wiki/Callback_(computer_science)) function when a specific event occurs on a [file descriptor](http://en.wikipedia.org/wiki/File_descriptor) or after a [timeout](http://en.wikipedia.org/wiki/Timeout_(telecommunication)) has been reached. Furthermore, libevent also supports callbacks due to [signals](http://en.wikipedia.org/wiki/Signal_(computing)) or regular timeouts.

SQSClient

Our SQSClient provides following APIs and their functions are:

* ***bool CreateQueue(String QueueName)*:** create a queue instance.
* *vector<String> ListQueues()*: list available queues.
* ***bool DeleteQueue(String QueueName)*:** delete a queue instance.
* ***bool SendMessage(String QueueName, String Message)*:** send a message to a queue. The size of the message can be up to 64 KB.
* ***String ReceiveMessage(String QueueName, int64 &MessageID):***receive a message from a queue. If called many times, all messages in the queue will be eventually received by a client. After the message has been retrieved, the client can process the message, which remains in the queue and is not returned to subsequent receive requests for the duration of the visibility timeout (usually in seconds).
* ***bool DeleteMessage(int64 MessageID):*** delete a message from a queue to avoid the message being received and processed again once the visibility timeout expires.

The client use the APIs above to communicate with master and get the datanode name and address (IP and port number), then it connect to the datanode and operate the command, if a datanode crushes, it will ask the master again and get another datanode’s address and request the command until one answers right. If no datanode can response correctly, the operation will return false.

And the detail design is like this



SQSMaster

Master is the main controller of the SQS system. It communicate with client and datanodes, and decides which datanode client should talk with.

Master has its own datacenter to log all the operations and datum. This datacenter store all the command the whole system operated. The datacenter get the message from nodes when they change some data and then dispatch this new change to all the alive datanode. Meanwhile it actively check all the datanode continuously to see if it still alive.

When a client comes, it responses the client with the datanode name and address(IP and port number) of one of the available datanode randomly. If client cannot get what she wants it will come back and the master should give the client another available datanode information.

The detail deign is like this



SQSDataNode

Datanode stores all the data in queue. For each system there are three Datanodes. If one of them crushes the others can replace this one to communicate with client and thus this mechanism maintains the availability of the whole system.

At the beginning, when a node is up, it will connect to the master and use recovery and join to participate in the SQS. It will receive the log and data from master. Then it waits the clients to communicate with it. It will give back the data or change the data as the clients want. When some data in its own node changes, the datanode will notify the master of the change and add log to record the change.

If a node crushes and is now up again, it will connect to master to get new information and update its log.

The detail design is like this



# Testing