# **Message queues for distributed systems**

**06 Jun 2016**

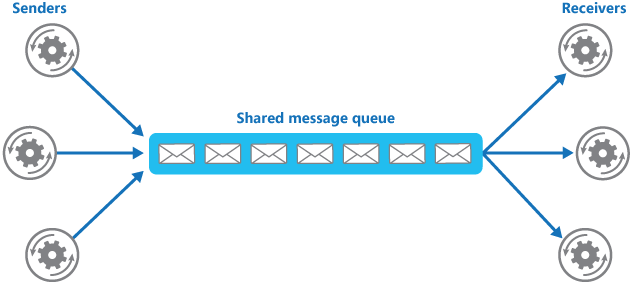
I’ve been working at [Simbiose Ventures](http://www.simbioseventures.com/) for the past six months in some projects that deals with huge amounts of data and that experience has driven me to always think on how the systems will scale in order to accommodate all incoming data without performance degrading as well. For the first time, I had to start thinking in [distributed systems](https://en.wikipedia.org/wiki/Distributed_computing). Soon I noticed that, when creating a distributed system, the first barrier we meet is to define some way to distribute the workload across many processes and computers. This article is on how we use message queues to achieve workload distribution and system decoupling.

The concept of message queues isn’t new in the computing world. In fact, I heard this name for the first time when studying Operating Systems at the university. Quoting the book [“Operating System Concepts”](http://www.amazon.com/Operating-System-Concepts-Abraham-Silberschatz/dp/0470128720), we can notice how message queues (a.k.a. message passing) was born for distributed computing.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

Let’s make things more concrete by giving an example. When I need to communicate with a friend, I just get my phone and send her a WhatsApp message or an e-mail so I don’t have to disturb the person at that exact moment. I can send as many messages as needed as they will be buffered in her phone until she is able to get it and read the messages, taking any action required. This is a message queue between two people communicating with each other, but the same pattern applies to distributed systems.

First, I have a process called **producer** that generates messages to be processed. Then, we have the **message queue** that stores messages in a **queue** - acting as a buffer - and route them to a **consumer**, some process that will process messages. Since a single queue is shared between virtually any number of producers and consumers, we can easily design systems to be scaled up quite easily by simply spawning more processes on more machines as far as they communicate through the same queue.



Queues provide asynchronous communication: each process controls its own flow and get messages when **they** are ready. As a consequence, message queues inherently enhance workload distribution given that computers will keep retrieving one more message and processing it until all incoming messages have been processed. Imagine the other scenario where we pre-assign messages to consumers before hand. It might happen that one process has finished processing all of its data whereas another one is struggling with its own workload. Since messages were assigned exclusively to the slow process, the quick one cannot jump over and help it finishing its job. We have idle resources even though there is work to be done. Point for message queues.

Need to create a fail-proof system that gracefully handles machine failures? Message queues can also help you with that. For instance, [Amazon SQS](https://aws.amazon.com/sqs/) has a [visibility timeout](http://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/AboutVT.html): a period during which the consumer need to process the message and delete it from the queue. In case the message isn’t deleted in viable time - such as when the process is stalled or has died - the message queue automatically routes the message to another consumer. You may even protect yourself from the machine hosting the message queue failing by using queues with replication capabilities, such as [Apache Kafka](https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Replication).

The last point I would like to write about message queues is about system decoupling. Queues are language-agnostic. A process written in Go can produce messages that will be consumed by a Python process without many troubles. Simply use the appropriate libraries and packages and you are ready to go. In fact, even different versions of the system can live happily side-by-side (but don’t give yourself headache without reason).

With these points in mind, message queues are my newest friends for designing and implementing distributed systems. I’ll leave you with some suggestions of message queue systems to take a look.

* [RabbitMQ](https://www.rabbitmq.com/)
* [ZeroMQ](http://zeromq.org/)
* [StormMQ](https://en.wikipedia.org/wiki/StormMQ)
* [Apache ActiveMQ](https://en.wikipedia.org/wiki/Apache_ActiveMQ)
* [Apache Kafka](http://kafka.apache.org/)
* [Amazon SQS](http://aws.amazon.com/documentation/sqs/)

Consider using one of them in your next project and please get in touch with me and share your thoughts on the experience.

### Benefits

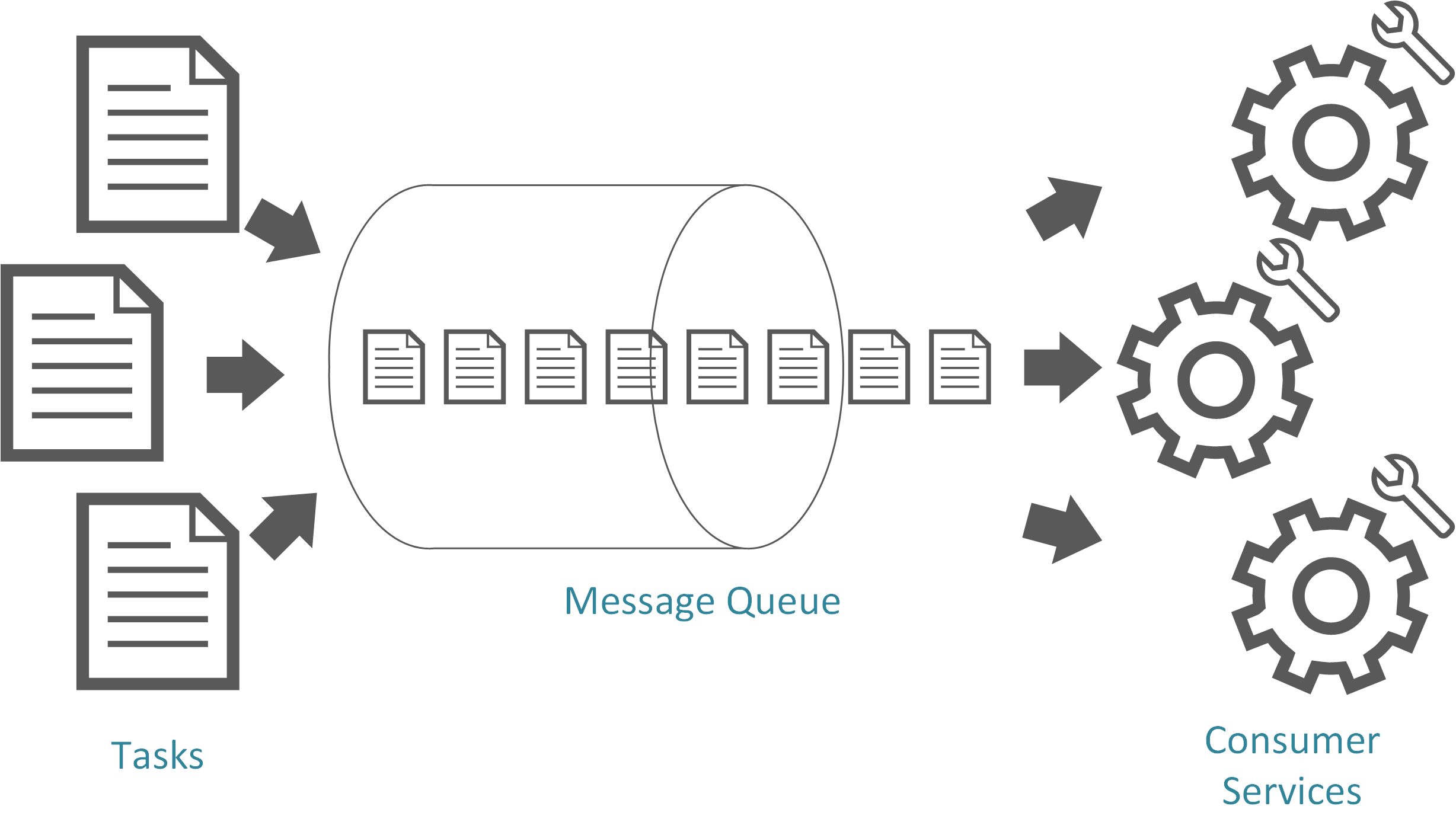
* Maximizing ****Availability****: Delays or Downtime of the service does not affect the application generating the messages
* Maximizing ****Scalability****: Number of Queues and Services can be varied to meet Demand. In [a later blog post, I discussed](https://blog.cdemi.io/design-patterns-competing-consumer-pattern/)**[the Competing Consumer Pattern](https://blog.cdemi.io/design-patterns-competing-consumer-pattern/)**, which helps in scaling your applications by running multiple instances of a service, each of which act as a message consumer from the load-leveling queue.
* Control ****Costs****: You don't have to design your service to meet peak load, but rather average load

### Considerations

The biggest consideration to keep it mind when implementing the Queue-Based Load Leveling Pattern is that ****an Asynchronous Queue is a one-way communication mechanism****! If a task expects a reply from a service, it may be necessary to implement a mechanism that the service can use to send a response.

## Competing Consumers

To handle this fluctuating workload, ****the system can run multiple instances of the consumer service****. However ****these consumers must be coordinated**** to ensure that each message is ****only delivered to a single consumer****. The workload also needs to be ****load balanced across consumers**** to prevent an instance from becoming a bottleneck.



### Benefits

* ****Scalability****: Depending on the backlog of the Queue, consumers can be increased or decreased on the fly.
* ****Load-Levelling****: Since this Design Pattern, inherently uses [the Queue-Based Load Leveling Pattern](https://blog.cdemi.io/design-patterns-queue-based-load-leveling-pattern/), all of the Benefits applied there apply here as well.
* ****Reliability****: If all the consumers are unresponsive or overloaded, the system can still continue working and no messages would be lost.
* ****Guaranteed Delivery****: At least once
* ****Resiliency****: If the consumer fails mid-task, the message is not lost, but rather returned to the queue, to be picked up by another consumer.

### Considerations

* ****Message Ordering****: Once you introduce multiple consumer instances, you are no longer guaranteed the order in which messages are received and the order in which they were pushed to the queue. Systems that use the Competing Consumer Pattern should ensure that message processing is idempotent.
* ****Poison Messages****: A malformed message, or any message that a service cannot process, may be returned to the queue crashing other services and creating and infinite loop processing these messages.
* ****One-Way****: As we discussed in [the Queue-Based Load Leveling Pattern](https://blog.cdemi.io/design-patterns-queue-based-load-leveling-pattern/), this Pattern is fully decoupled, so if the service generates a result, it must be passed back to the application logic and must be stored in a location that is accessible to both.