Decoupling Distributed Systems: Solving Bottlenecks with Message Queuing

A Seminar Report

Submitted on partial fulfillment of the requirements for the award of the degree of Master of Computer Applications under National Institute of Technology, Karnataka by

Apurv Kumar
Roll No: 244CA008

Under the Guidance of
Prof. Santhosh George
Dr. Gayathri P

DEPARTMENT OF MATHEMATICAL AND COMPUTATIONAL SCIENCE

February - 2026

DECLARATION

I hereby declare that the seminar report entitled “Decoupling Distributed Systems: Solving Bottlenecks with Message Queuing” which is being submitted to the National Institute of Technology Karnataka, Surathkal, in partial fulfillment of the requirements for Mandatory Learning Course (MLC) of Master of Computer Applications in the Department of Mathematical and Computational Sciences, is a Bonafide report of the work prepared by me. This material is collected from various sources with utmost care and is based on facts and truth.

Apurv Kumar
244CA008
MCA 4th Semester

NITK, Surathkal

CERTIFICATE

This is to certify that the P.G. seminar report entitled “Decoupling Distributed Systems: Solving Bottlenecks with Message Queuing” submitted by ‘Apurv Kumar’ (Roll number: 244CA008, Registration number: 2440058) as the record of the work carried out by him is accepted as the P.G. Seminar report submission in partial fulfillment of the requirements for the mandatory learning course of Master of Computer Application in the Department of Mathematical And Computational Sciences, NITK, Surathkal.

Faculty Guide

Dr. Santhosh George
Department of MACS NITK, Surathkal

Dr. Gayathri P
Department of MACS NITK, Surathkal

ACKNOWLEDGEMENT

I am writing to express my sincere gratitude to all those who have con tributed, directly or indirectly, towards completing this seminar report entitled “Decoupling Distributed Systems: Solving Bottlenecks with Message Queuing”. First and foremost, I would like to thank my seminar guide, Dr. Gayathri P and Dr. Santhosh George for their valuable guidance, encouragement, and constructive suggestions throughout the preparation of this report. I am deeply grateful for the time, patience, and constant support extended during this work. I would also like to extend my heartfelt thanks to Dr. Pushparaj Shetty D, Head of the Department of Mathematical and Computational Sciences, NITK Surathkal, for providing me with the opportunity, facilities, and academic environment to undertake this seminar as part of the Mandatory Learning Course. I am thankful to all the department faculty members for their insightful lectures, resources, and academic inputs, which greatly helped shape my understanding of the topic. I would also like to acknowledge the library and digital resources of NITK, which provided access to several references that were instrumental in preparing this report.

Apurv Kumar

244CA008

MCA 4th Semester

NITK, Surathkal

ABSTRACT

Distributed systems are widely used in modern software applications due to their ability to support scalability, flexibility, and modular development. However, such systems often suffer from performance bottlenecks and reliability issues because of tight coupling and synchronous communication between components. Performance bottlenecks can lead to increased response times, degraded user experience, and cascading failures that affect system availability. Message queuing is an effective architectural approach that helps in decoupling system components and enabling asynchronous communication, allowing services to operate independently and continue processing even during high traffic or partial failures. This report presents a detailed study of how message queuing helps in solving bottlenecks in distributed systems. It discusses the fundamentals of distributed systems, common bottlenecks, core concepts of message queuing, architectural design, real-world applications, advantages, limitations, and key design considerations. The objective of this report is to provide a clear and practical understanding of how message queuing supports the design of efficient, robust, and reliable distributed systems, with insights relevant for both academic study and real-world system design.

TABLE OF CONTENTS

1. Introduction

Distributed systems consist of multiple independent components that work together to achieve a common goal. These components usually run on different machines and communicate over a network. With the growth of cloud computing, microservices, and large-scale web applications, distributed systems have become the standard approach for software development.

1.1 Need for Decoupling

In tightly coupled systems, components depend directly on each other for execution and availability. This dependency leads to several issues:

- Reduced flexibility in system design
- Poor fault tolerance when one component fails
- Difficulty in scaling individual services

Decoupling allows components to operate independently, making the system easier to scale, maintain, and evolve.

1.2 Role of Message Queuing

Message queuing enables asynchronous communication between components. Instead of waiting for immediate responses, services exchange messages through a queue. This approach reduces direct dependency between services and improves overall system performance.

2. Overview of Distributed Systems

A distributed system is a collection of autonomous computers or services that communicate and coordinate through message passing. Each component performs a specific task and collaborates with others to provide complete system functionality.

2.1 Characteristics of Distributed Systems

Distributed systems are defined by a few fundamental characteristics. Resources such as data, computation, and services are shared among multiple components. Operations occur concurrently, which improves performance but also introduces coordination challenges. There is no global clock to perfectly synchronize all components, and each component can fail independently without immediately stopping the entire system.

2.2 Challenges in Distributed Systems

Despite their advantages, distributed systems face several practical challenges. Network latency and communication failures can slow down interactions between services. Partial failures are common, where one component fails while others continue to operate. Maintaining data consistency across services is difficult due to concurrent access, and scaling tightly coupled services often increases complexity and cost. These issues frequently result in performance degradation and reduced reliability.

3. Bottlenecks in Distributed Systems

A bottleneck is a point in the system where performance is limited by a single component or shared resource. In distributed systems, bottlenecks commonly arise due to excessive dependencies between services.

3.1 Causes of Bottlenecks

Bottlenecks in distributed systems usually arise from design choices that create strong dependencies between services. Synchronous service-to-service communication forces one component to wait for another to respond. Shared databases or centralized resources become points of contention under high load. Uneven workload distribution further increases pressure on specific components, while repeated retries during failures amplify system stress.

3.2 Impact of Bottlenecks

Bottlenecks can lead to:

- Increased response time
- Reduced system throughput
- Cascading failures, where the failure of one service affects multiple dependent services.

4. Message Queuing Fundamentals

Message queuing is a communication method in which messages are exchanged between producers and consumers through an intermediate queue. The queue temporarily stores messages until they are processed.

4.1 Basic Components

A message queuing system consists of several logical components working together. Producers generate messages and send them to a queue. The queue temporarily stores these messages until they are processed. Consumers retrieve messages from the queue and perform the required operations. A message broker manages the entire process by handling message storage, routing, and delivery acknowledgements.

4.2 Message Flow

Figure 1: Producer → Message Queue → Consumer flow

The producer places messages into the queue without waiting for processing. Consumers retrieve messages from the queue when they are ready to process them.

5. Architecture of Message Queuing Systems

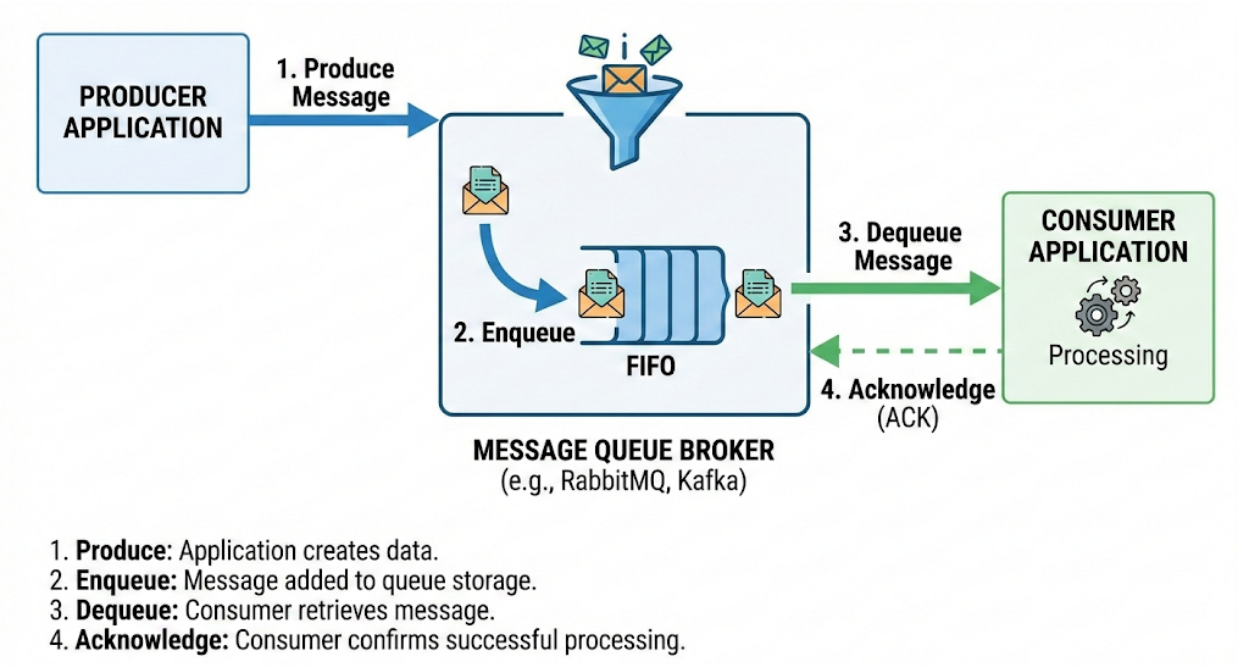

Figure 2: Broker-based message queuing architecture showing producers, message queue, and consumers

Message queuing systems follow a broker-based architecture that clearly separates message producers from message consumers. This separation is fundamental in eliminating direct dependencies between services and forms the backbone of decoupled system design.

In this architecture, producers are responsible for generating messages and sending them into the queue. They do so asynchronously, meaning that they do not wait for the consumer to process the message before continuing their own work. This contributes to better system throughput and avoids unnecessary blocking of processes.

Consumers, on the other hand, read messages from the queue when they are ready to process them. They can be scaled independently based on load. For example, if incoming messages grow rapidly, more consumer instances can be added to handle the backlog without changing how the producer operates.

The message broker manages the entire process. It ensures that messages are stored reliably, that they are routed correctly to consumers, and that delivery acknowledgements are handled. The broker also tracks whether messages have been processed so that they are not lost even if parts of the system fail temporarily.

This architecture allows developers to design systems where services remain autonomous, isolated from failures in other parts of the system, and able to scale independently. It also lays the foundation for more resilient and maintainable distributed systems.

6. Asynchronous Communication Model

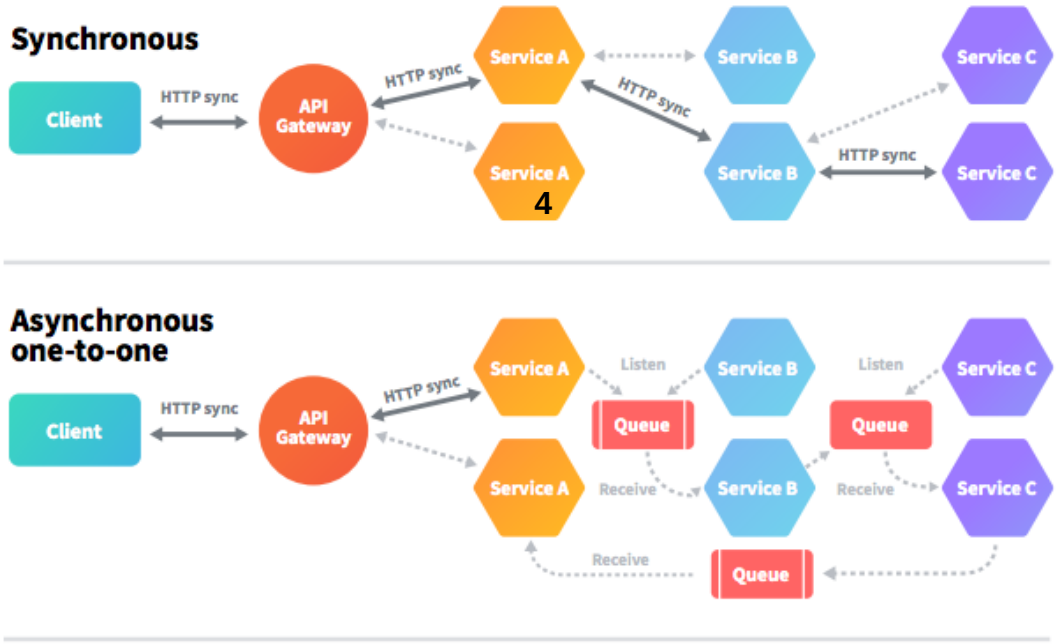

Figure 3: Synchronous vs asynchronous communication.

Asynchronous communication is a communication pattern where services do not wait for responses before proceeding to other tasks. Unlike synchronous communication, where a request blocks until a reply is received, asynchronous models allow services to continue execution, improving responsiveness and system utilization.

In a synchronous model, a service must wait for a response before continuing. This can become problematic under heavy load. For example, if a front-end service must wait for a payment service to confirm a payment before proceeding, the entire request processing pipeline can slow down if the payment service is overloaded.

In contrast, asynchronous communication decouples the sender and receiver in time. A service can send a request or message to a queue and move on to other work. The receiver service picks up that message later whenever it is ready. This leads to better utilization of resources and improved responsiveness, especially in systems experiencing variable workloads.

One direct benefit of asynchronous processing is better system elasticity: resources can be added or removed based on processing load without disrupting ongoing processes. It also increases fault tolerance, because services are not tightly bound to the immediate availability of others, and messages can be persisted until they can be processed safely.

7. Role of Message Queues in Solving Bottlenecks

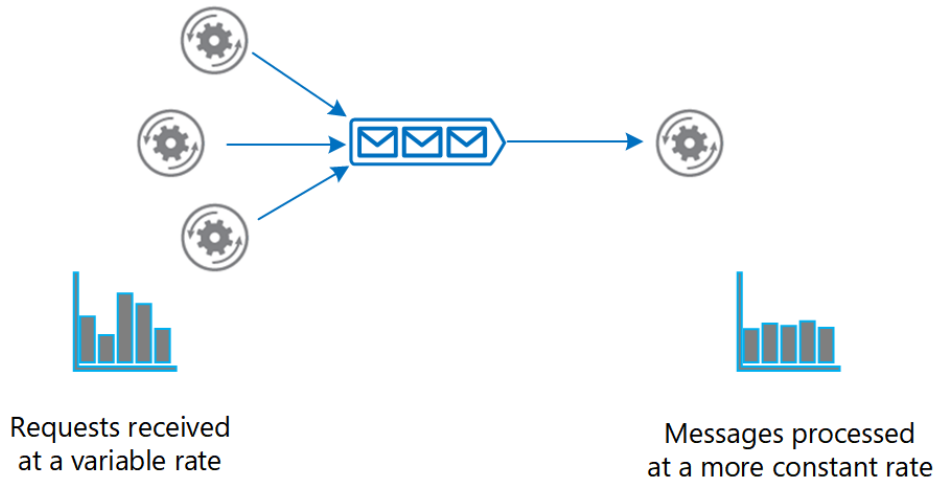

Figure 4: Load buffering mechanism where message queues absorb traffic spikes and prevent service overload.

Message queues play a crucial role in reducing bottlenecks in distributed systems by enabling asynchronous and buffered communication between components. When services send messages to a queue instead of directly to each other, the system gains flexibility in how requests are processed.

One major way message queues address bottlenecks is through **load buffering**. When a large number of requests arrive at once, the queue stores them and allows consumers to process them at a pace they can manage. This prevents services from becoming overwhelmed during peak loads and helps maintain system stability.

Message queues also improve **fault tolerance**. If a consumer fails during processing, the unprocessed messages remain in the queue. When the consumer recovers, it can continue processing without losing messages or causing data inconsistencies. This behavior significantly improves reliability because temporary failures in one component do not cause the entire system to fail.

Another important aspect is **background processing** of tasks that are not immediately required for a user's response. For example, sending an email confirmation or generating logs can be offloaded to the queue, allowing the main processing to respond quickly to the end user.

By buffering load, isolating failures, and enabling background work, message queues reduce points of congestion and improve the overall responsiveness and reliability of distributed systems.

8. Popular Message Queue Technologies

Several message queue technologies are widely used in industry.

8.1 RabbitMQ

- Supports multiple messaging protocols.
- Widely used in enterprise applications.

8.2 Apache Kafka

- Designed for high throughput systems.
- Commonly used for event streaming and data pipelines.

8.3 Amazon SQS

- Fully managed cloud-based message queue service.
- Automatically scales with demand.

9. Real-World Use Cases

Message queues are widely used in practical applications.

9.1 E-Commerce Applications

- Order processing
- Inventory updates
- Payment notifications

9.2 Logging and Monitoring Systems

- Centralized log collection
- Event-driven monitoring and alerts

9.3 Email and Notification Services

- Asynchronous message delivery
- Improved reliability and fault tolerance

10. Design Considerations

Designing a message queue-based system requires careful planning. Messages should be durable enough to survive failures when required. The system must clearly define delivery guarantees such as **at-least-once** or **exactly-once** delivery. Message ordering can be important for certain use cases, and proper error handling with controlled retries is necessary to avoid message loss or overload.

11. Advantages and Limitations

11.1 Advantages of Message Queuing

- Loose coupling of services
- Improved scalability
- Enhanced fault tolerance
- Efficient handling of variable workloads

11.2 Limitations of Message Queuing

- Increased system complexity
- Debugging challenges in asynchronous systems
- Eventual consistency issues

12. Best Practices

The following best practices help in effective use of message queues:

- Monitoring queue length and processing latency.
- Designing idempotent consumers.
- Implementing proper retry and error handling mechanisms.

13. Future Scope

The role of message queuing will continue to grow significantly as distributed systems evolve. With the increasing adoption of cloud-native application design and microservices architecture, there is a trend toward event-driven systems that rely heavily on asynchronous message processing. In such architectures, decoupling through queues allows services to remain independent and scalable, which is considered a best practice in modern backend development.

Integration of message queues with serverless platforms like AWS Lambda, Google Cloud Functions, and Azure Functions is another area with growing importance. Such integration enables developers to build highly scalable, event-driven applications without worrying about infrastructure provisioning, allowing systems to scale automatically based on load.

Another promising direction is the enhancement of observability and monitoring tools specific to message queue systems. This includes real-time dashboards, automated anomaly detection, and performance tracing that help developers ensure reliability and optimize throughput.

Finally, research in improving delivery guarantees, such as exactly-once message processing and transactional messaging across distributed components, will further strengthen the reliability of message queuing, making it an important area for both research and practical investment.

14. Conclusion

Message queuing is a highly effective architectural approach for decoupling services in distributed systems and addressing performance bottlenecks. By enabling asynchronous communication and buffering of messages, it minimizes direct dependencies between components, allows independent scaling of services, and increases fault tolerance. This improves system flexibility and enables better handling of variable workloads, which is critical in large-scale and cloud-based applications. Studying message queuing highlights its importance not just as a technical mechanism, but as a core design pattern for robust, efficient, and responsive distributed systems. When implemented thoughtfully, message queuing supports scalable architectures that are easier to maintain and extend over time.

References

Textbook

[1] Tanenbaum, A. S., *Distributed Systems: Principles and Paradigms*
[2] Kleppmann, M., *Designing Data-Intensive Applications*

Web Resources

[1] *RabbitMQ Official Documentation,*
https://www.rabbitmq.com/documentation.html

[2] *Apache Kafka Documentation,*
https://kafka.apache.org/documentation/

[3] *CloudAMQP Blog, What is Message Queuing,*
https://www.cloudamqp.com/blog/what-is-message-queuing.html

[4] *Google Cloud Documentation, Event-Driven Architectures,*

https://cloud.google.com/eventarc/docs/event-driven-architectures

[5] *Microsoft Azure Documentation, Message Queue Overview,*

https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview

[6] *IBM Documentation, Messaging and Event Streaming,*
https://www.ibm.com/docs/en/event-streams